Process query systems : advanced technologies for process detection and tracking

Berk, V.H.

Process Query Systems

Advanced Technologies for Process Detection and Tracking

Proefschrift

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van de Rector Magnificus Dr. D. D. Breimer,
hoogleraar in de faculteit der Wiskunde en
Natuurwetenschappen en die der Geneeskunde,
volgens besluit van het College voor Promoties
te verdedigen op woensdag 18 januari 2006
klokke 15:15 uur

door

Vincent Hendrik Berk

geboren te Leidschendam in 1978

Promotiecommissie

Promotores:	Prof. dr. H.A.G. Wijshoff	
	Prof. dr. G.V. Cybenko	(Dartmouth College, USA)
Referent:	Prof. dr. W. Jalby	(Université De Versailles, France)
Overige leden:	Prof. dr. S.M. Verduyn Lunel	
	Prof. dr. F.J. Peters	
	Prof. dr. ir. E.F.A. Deprettere	

Parts of this research were supported by grants from the US Department of Justice, and the US Intelligence Community (award numbers: DOJ-FBI-NIPC A3N308035 (9/04–9/05), IC-ARDA-P2INGS F30602-03-C-2348 (10/03–3/05), and DOJ-ISTS 2000-DT-CX-K001 (12/03–11/04)).

To my parents

Contents

Preface

In this dissertation I present the results of my research in Process Query Systems. This work started out four years ago as a search for methods capable of processing the enormous amounts of streaming data that is generated by an Internet-scale worm detection system. The design of this system, named DIB:S, was started at Dartmouth College and soon became a reliable and early indicator of active worms. In an effort to generalize the detection concept, the first principles of Process Query Systems were developed. By modeling interesting events as processes, many problems can be described in new and powerful ways. These processes are then submitted as queries to the PQS, which then returns evidence of these processes occurring in observed event streams.

As it turns out, Process Query Systems can be used for many different tasks, including kinematic tracking, autonomic server monitoring and control, covert channel detection, and the monitoring of security on large, enterprise-class computer networks. By simply expressing the relevant processes as PQML (Process Query Modeling Language) models, a PQS can quickly be applied to any given domain. This dissertation gives an in-depth description of the PQS concept, algorithms, and performance analysis, as well as providing a number of very diverse examples using PQS, demonstrating its wide applicability.

Although a comprehensive study of Process Query Systems is offered, the reader must realize that the area is new, and many applications remain to be studied. I aimed to make the majority of this work comprehensible to the general public, however, some fundamental knowledge of computer science, mathematics, and network security is helpful in understanding all the aspects of PQS.

Finally, I am indebted to many people who were instrumental to the development of Process Query Systems, and I would like to thank them for all their help. Especially, I would like to thank Marion Bates for her tedious proof-reading of the manuscript. And I would like to thank my parents for their endless support.

Chapter 1

Introduction

1.1 Motivation

This thesis introduces a new family of algorithms and software implementations that solve a broad class of problems classified as process detection. These algorithms are called Process Query Systems. Several applications and examples are introduced that demonstrate the power and applicability of PQS.

During the development of an Internet-wide worm tracking and detection system we quickly realized that the number of events that need correlation rises far beyond the capacity of already existing approaches. A fast moving worm will initially saturate any specific-purpose detector, and then cause such significant Internet instabilities that entire sensor arrays will become unreachable. Due to this scalability problem accurate detection must happen within the first few seconds after the release of the worm.

Large scale data processing for detection quickly grew to the broader task of building a generic tracking system. Tracking implies a deeper understanding of the subject than detection. Detection simply says: "there is a *THING*", whereas tracking actually tells you: "the *THING*, which was detected, is going to do *FOOBAR*", anticipating what may occur. The objective therefore was: "can we decide, seconds after a worm is released, how big, how fast, and how destructive it will be?" We realized that we needed a general approach that recognizes the dynamics of an observed system, rather then merely the signature of its existence. This meant finding evidence of an occurring *Process* in a stream of observed events: a Process Query System solves this problem.

During the development of an initial PQS implementation, many questions were raised and many lessons were learned. This work by no means answers all these questions, although it introduces most of them. It will likely take many more years for Process Query Systems to mature and reach main-stream computing, even though a commercial application is already nearing completion. Most importantly, there seems to be no end to the applicability and the power of the paradigm; the same PQS kernel has already been applied to: Worm detection, kinematic vehicle tracking, enterprise-class network security monitoring, covert channel detection, server farm monitoring, and the kinematic tracking of fish in a tank. All these applications are very diverse in nature and show the generically applicable power of process detection.

1.2 Background

The field of *Random Signal Processing* deals with natural events producing noisy and lossy signals. These natural events are usually modeled by *Stochastic Processes* occurring in the observed environment. Typically, signals are also referred to as observations or sensor reports. Natural signals generally exhibit some inaccuracy (like spatial deviation), false positives (an observation while there should have been none), false negatives (no observation while there should have been an observation), duplicates (two observations for a single instance), overlap (one instance obscures the other), etc. All these factors make proper process estimation a difficult task. The goal is to properly estimate the state

of the observed environment, as accurately as possible. This often involves some type of *hidden states*, meaning that the state of the environment is not directly observed by the sensors, and that the actual state has to be derived from what the sensors are picking up.

The following section outlines the problem definition and explores some of the terminology that is used throughout this thesis. Although the problem definition is broad, we believe the solution offered here is able to fully address this need. Later on we give an exploration of some of the classical methodologies in the field, and how they compare to the proposed general solution.

1.2.1 Problem definition

We start by giving the following problem definition, and then continue by exploring its separate parts:

> *The need for a generic, generally applicable framework for disambiguation, detection, tracking, and state prediction of multiple discreet and/or continuous stochastic processes in noisy and lossy environments.*

What is a stochastic process?

Formally, a stochastic process is a collection of random variables $X(\omega,t)$ defined on a given probability space, indexed by time variable $t \in T$, where $\omega \in \Omega$ is the sample space representing the state of the process [122, 21]. Intuitively, there is some mapping function between any t and the expected outcome o_t (where o_t is the expected observable). This mapping function changes for different values of t. Additionally, in general the expected outcome o_t at time t depends on the one or more observables before t, thus creating a state sequence.

Both the sample time and the observable values can be random or discreet. An example of a discrete time, continuous value stochastic process is the measurement of engine temperature in a car. The car computer will measure the temperature at fixed times, for example once per second, however, the temperature itself is a continuous value. An example of a continuous time, discrete value stochastic process is a queue at the grocery store. At any given time there are 0, 1, 2, 3, ... people in line, never $2\frac{1}{2}$. However, the arrival time of new customers and the moment that the attendant finishes with a customer are continuous. The size of the queue is not a simple random variable since the attendant can only help one customer at once. This means that the size of the queue at t depends on the size of the queue before time t.

The methods for modeling stochastic processes are the same for continuous or discrete time, whereas they differ for discrete and continuous values. Therefore time is often simply taken to be continuous. Discrete value stochastic processes can for example be modeled by Markov chains [72], continuous stochastic processes can often be modeled by Wiener Filters [123] such as the Kalman Filter [58].

Furthermore, because future observable events generally depend on the foregoing sequence of observable events, stochastic processes are considered stateful. The state of the process at time t is therefore defined by the sequence of observables leading up to time

t, although the actual state is not necessarily defined as *the sequence of known observations*. This means that the actual state of the process may not always be readily apparent from the observed events. Such processes may be modeled with Hidden Markov Models [99].

What is meant by noisy and lossy?

Noisy and lossy mostly concerns the sensors and the environment that are used to obtain the observables. Consider for instance the case of tracking airplanes around an airport using a radar. The "blips" that are returned will certainly be noisy, because of birds flying close to the airport, bad weather conditions, radar reflections, etc. Also, the blips will appear different depending on the angle that the plane makes with the radar waves. We may miss radar observations when a plane goes behind a hill, or is obscured by another plane. For some sensors we will know the effect of the environment on its accuracy, for other types of sensors only a rough estimate can be made.

It is therefore hard to predict what the radar will actually see when we know that a plane is flying through the airspace (the actual stochastic process). Additionally, the full state of the plane is not directly observable by the radar, if we consider the plane's direction and speed as part of its state. The radar will only observe the estimated position. A Kalman filter will help estimate the actual state of the process, and the error in this estimation, thus aiding in the interpretation of the radar observations.

What about multiple processes?

This is where the complexity of the problem explodes. Consider the same airspace from the example above, but now we put 10 planes in it. For every sweep of the radar we expect to see about 10 blips, one for each plane, plus an undetermined number of birds, reflections and other noise. How do we *generically* decide which blips belong to what process? What can be considered noise? Which blips signal the arrival of a new plane? The ultimate goal is to detect occurring processes by examining their behavior (i.e. their observables), however, this cannot be done if the observables from multiple processes are intermixed. The system must therefore offer some way in which the observations are disambiguated before an attempt is made to determine what process is occurring. This disambiguation is also referred to as "event gating".

In real world environments, however, the observed events often come from several *different* processes. Consider in the radar example that propeller planes will behave differently on radar than passenger jets, although the blips they generate are exactly the same. Only by correlating multiple series of radar blips and analyzing their behavior can we determine if we are dealing with a jet plane or a propeller plane. The multiple processes component will therefore get even more complicated if not all event producing processes are of the same type.

Detection, tracking, and prediction

So far the discussion has focussed on the environment: a complex space in which multiple instances of one or more different stochastic processes are generating observable events. These observable events come out of the environment by means of noisy and

lossy sensors, or sensor networks. The ultimate goal is, based only on the received observables, to detect which processes are occurring and what state they are in. Tracking implies a deeper understanding of the process than detection alone: it implies that we understand the changes of state that the process went through to generate the observed sequence of events. This actually helps in observation disambiguation, since tracking of state changes also allows us to formulate an expected *next* observable: i.e. a prediction.

Predictions are usually a stochastic function specific to the (near) future time slices t, which basically implies a probability distribution on several possible expected observables. These predictions make for improved accuracy in observation disambiguation, meaning that new incoming observations can more easily be matched with already detected processes that are now being tracked. Events that do not directly correlate may therefore be considered the start of a new process in the environment, or the effects of noise. An assignment of observed events to tracked process instances is usually referred to as a *hypothesis*. An hypothesis is considered to be "internally consistent" when no observed event is assigned to more than one tracked process.

Generally applicable framework

Although most of the above problems have been solved, or partially solved, for very specific cases, no generally applicable framework exists. Most, if not all, solutions duplicate a lot of basic functionality, such as observation handling logic, observation disambiguation and pairing, hypothesis generation, and hypothesis pruning. This basic functionality calls for a generic framework in which new stochastic processes can easily be modeled, such that a detection, tracking, and prediction system can quickly be built for any specific area.

This idea calls for a clear separation of what is taken care of by the framework and what is considered part of the modeled processes. Also, it is very important to set a clear standard on how to model stochastic processes as generically as possible, to allow for application in many specific domains. Process models should thus be considered *domain-specific plug-ins*. The framework system is thus applied to a specific domain by inserting the process models. All other logic in the framework should be domain-independent.

Finally, no framework is generic if it cannot accommodate most of the existing *case-specific* approaches. It is therefore required that most, if not all, of these existing approaches can fit right in and be easily implemented using the general framework. A system that meets all the above requirements and satisfies the defined need is called a "Process Query System".

1.2.2 The Discrete Cause Separation Problem

This section formalizes the above problem definition further and introduces an example that is used in the next section to compare existing approaches.

As discussed above, observable events are produced by one or more given *processes* occurring in an environment. For example, an airplane flying through the sky is a process, and the observable events could be a stream of radar observations (i.e. x,y,z-coordinates).

This implies that *processes* are the cause of the observed event stream. The process changes state either continually (the position of the airplane) or discretely (the block of airspace the plane is currently in), and the observed events will be a representation of that state, with a given error. However, due to the limitations of most sensing systems, it is often not directly clear what process produced the observed events. Consider two airplanes in the same airspace, for instance, then the radar will report on two x,y,z-coordinate tuples per pass.

Other examples of process-driven applications include network security monitoring, where scanning and intrusion processes trigger network based sensors. In network management, the processes that occur are service failures caused by buggy software, or failures of hardware. Military situational awareness typically involves the tracking of troop and vehicle movements, all modeled by processes. Although these applications may seem very different in nature, they actually share many common features when viewed from an appropriately abstract perspective. This abstract framework concerns a collection of processes $\{\mathcal{M}_1, \mathcal{M}_2, \ldots\}$, which is producing a stream of observable events:

$$\ldots, e_i, e_{i+1}, e_{i+2}, \ldots$$

where event e_j occurs at time t_j where $t_j \leq t_{j+1}$. The goal in these applications is to solve the inverse problem: determine which instances of which processes produced the observed sequence, keeping in mind that some processes have discrete states, while others have a continuous state. A PQS is a software system that solves this class of problems.

It must be noted early on that the sequence of observed events is usually not perfectly chronological, nor is it often complete. In practical environments the observable events are not always time-stamped at the time of detection, meaning that the only time-stamp available is when the observed event arrives at the PQS. Additionally, it is not uncommon for observations to arrive out-of-order, or for observations to be lost entirely. Finally, it is unknown what number of processes is causing the observed event stream, meaning that multiple instances of a single process may be responsible for the events (for example: Are there one, two, or three airplanes in radar range?). As will become clear from the following discussion, these fundamental aspects of the problem are the reason that previous algorithmic and software approaches are poorly suited for the process detection task.

The class of processes can be diverse and processes often contain "internal" or "hidden" states, which are not directly externally observable. These hidden states produce observable events from which the existence of the process must be determined as observations arrive. Process Query Systems solve in the broadest sense what we refer to as the *Discrete Cause Separation Problem* (DCSP). The DCSP is informally stated as follows:

> *The Discrete Cause Separation Problem* (DCSP) - Given a finite sequence of observed events, captured at times $t_1, t_2, \ldots, t_n$,
>
> $$e_{t_1}, e_{t_2}, \ldots, e_{t_n}$$
>
> and a collection of processes, $\{\mathcal{M}_1, \mathcal{M}_2, \ldots\}$, determine:

1. The "best" assignment of events to process instances, namely

$$f : 1,2,\ldots,n \rightarrow \mathbb{N}^+ \times \mathbb{N}^+,$$

   where $f(i) = (j,k)$ is interpreted as meaning that event e_i at t_i was caused by the kth instance of process model $\mathcal{M}_j$ (the process detection problem);

2. The corresponding internal states and state sequences of the processes thus detected (the state estimation problem).

Here $\mathbb{N}^+ = 1,2,\ldots$ is the set of positive integers. (The name, Discrete Cause Separation Problem, is by analogy with the Blind Source Separation Problem that arises in continuous signal processing [54].)

This broad description of DCSP intentionally omits many details. The "best" assignment is ultimately determined by an application-specific scoring function between sequences of events and process models. Note also that this "best" assignment is not necessarily a perfect assignment that explains all observed events. Even if such an assignment were possible, it may not always be computable. In a practical environment there will be noise, lost observations, and a number of other, unknown processes occurring, which complicate the assignment of process models to events. Also, given a stream of observable events, and a set of process models, the number of possible assignments grows exponentially, meaning that any *practical* solution to the DCSP will be forced to have data reduction steps. These data reduction steps will inevitably lead to a loss of information as a set of possible assignment options. This implies that any practical implementation of of a PQS will be a heuristic method: The "best" assignment is not necessarily always the perfect one, although usually it will be close.

Additionally, in order to have a software system make an assignment between observable events and processes, the causal relationship between processes and events needs to be made explicit. This leads to the central question of how processes are described and how process models are created in the first place. The central premise is that there is no one-to-one relationship between observed events and states in a model, as this would trivialize the problem. Accordingly, we will assume that different processes and different states of the same process instance can give rise to the same observable events. This means that temporal and causal relationships between states within a process must be exploited to disambiguate between states and various processes. For instance, although any airplane may be able to trigger radar observations all over the world, if two radar observations are received from opposite ends of the globe, within several seconds of each other, they will not have been triggered by one and the same airplane.

1.2.3 Example

Consider the following example of a multi-process DCSP, containing two different models, each of which trigger exactly the same observations (i.e. an x,y,z-coordinate tuple). Assume the first model describes the dynamics of a propeller plane. We will refer to

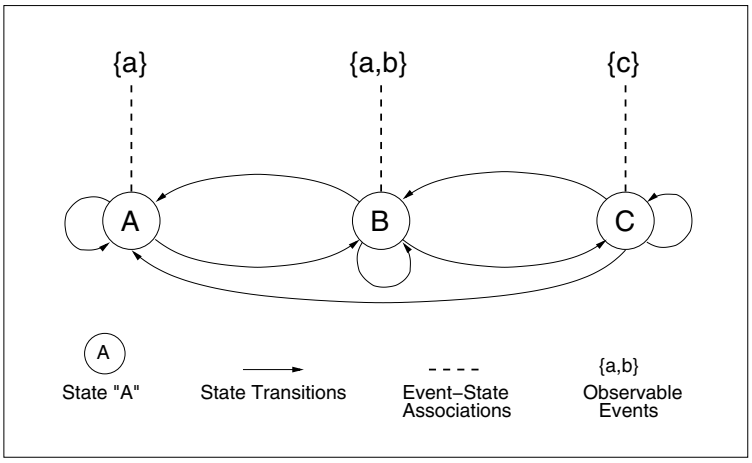


Figure 1.1: A Simple Process Model, $\mathcal{M}_1$

this model as $\mathcal{M}_1$. Then the second model $\mathcal{M}_2$ describes the dynamics of a fighter jet, both observed by radar. Since both types of plane generate the same type of radar observation the disambiguation must focus on the specific dynamics of the different planes. A jet fighter will generally fly much faster, and turn much faster than a propeller plane. Also, propeller planes are less likely to fly in groups. The models would reflect these dynamics. Now, given an actual environment, it may very well be possible that there are several propeller planes and a group of jet fighters in the same airspace, all within radar range. The PQS will use the radar data as input observations together with the two models to disambiguate which radar observations were triggered by which aircraft by associating radar observations using the models. Subsequently, the hypothesis will be that there are several instances of the model $\mathcal{M}_1$ (the propeller plane) and a group of instances of model $\mathcal{M}_2$ in the observed environment.

This is a very complicated problem, and before discussing the more general problem of multi-process DCSPs, we first address single process DCSPs. Generally speaking, a process is defined by a set of states and a description of the dynamics that define state evolution over time. That is, at any instant in time, the process occupies a state and at some future time transitions to another state. The question of whether such transitions are clocked (synchronous) will not be addressed, however synchronous state transitions tend to simplify the problem since we know when to expect the next observable. We do, however, use the traditional Markovian definition of "state", in that the dynamics of future state evolutions depend only on the current state. This limitation is taken initially to clarify the approach, however there is no fundamental reason that constrains a PQS to 1^{st} order Markovian models only. Additionally, although discrete states are used in the following examples, states do not necessarily have to be discrete. For example, the state of a moving vehicle would be its current position and speed, and possibly its acceleration.

Given is a process, $\mathcal{M}_1$, as depicted in Figure 1.1. $\mathcal{M}_1$ has three states, A, B, C. The

arrows between states indicate the possible transitions. So we can go from state A to state B, but not directly from A to C. These three states and their transitions from a classical Finite State Machine [125]. In addition to the possible transitions, each state also has an associated set of possible observable events that are characteristic for the process. These observable events are represented by the dotted line. This means that a process modeled by $\mathcal{M}_1$ produces observable events a when it is in state A, and produces both a and b events when it is in state B, and so on.

As an example, assume that the process $\mathcal{M}_1$ models the operation of a network device, where state A is "normal" operation, state B is "abnormal" operation and state C is "failure". Observable event a would be a normal response to a Simple Network Management Protocol (SNMP) query, event b would be an abnormal response (an error code, for example), and event c would correspond to no response at all. Note that the no response observable, namely γ, could be due to a network link failure as well as a device failure. A link failure could be modeled by a process similar to $\mathcal{M}_1$, but with different states and similar, or different observables. Likewise, model $\mathcal{M}_1$ could depict a process in an entirely different domain, like military target tracking, or interstate traffic control. The central point is that in many application areas the need arises for a generic modeling approach. Although this example is very simple, it shows the generality of the PQS approach.

Assume that the observable events are sensed and delivered to the tracking system by a sensor network system. Such a sensor network system often contains many sources and is designed to route and possibly pre-process the data before it is handed to the PQS system. Note also that such a sensor network system will generally not be devoted to collecting events for just one process. A sensor network system will collect and pre-process observable events for the whole of the observed system and deliver them to the PQS core. Likewise the PQS core will have many process models, each describing one possible process that can take place within the observed domain. Let us look at an example to get a better sense of what this means. The engine of a car has hundreds of internal sensors, all providing a constant stream of data, regarding fuel mixture, combustion information, temperature, engine speed, and so on. The sensor network system in this case would be those sensors, and the network of wiring that gets all the sensor data to the central car computer. Some sensor data might need some pre-processing before it is meaningful, this can be done in-line. The central car computer will be running a PQS with many different process models, looking for evidence of improper engine functioning. The models can be as simple as looking for retarded ignition timing, and as complex as the detection of excessive piston ring wear.

Getting back to the original example of one model $\mathcal{M}_1$, note that three observed events, namely a, b, and c, are modeled as being associated with this process model. Let x be any event not associated with this process (or more specifically: any event not modeled to be associated with this process). In the DCSP framework, we observe the following event sequence:

$$e_1 = a, e_2 = c, e_3 = b, e_4 = a, e_5 = c$$

which we shall denote as $e_{1:5} = acbac$ for convenience. The DCSP in this case is defined as the state sequence that best accounts for the sequence of observed events. An example of a possible state sequence, given model $\mathcal{M}_1$ is *BCBBC*, while *ACBAC* would not be because there are no transitions directly from state *A* to state *C*. A straightforward way of matching a possible state sequence to a given observation sequence is to compare the observation sequence to a list of all possible state sequences. Such a list is easily generated iteratively by starting out in all the possible starting states of a model, and then growing a tree for all reachable states in the next step and so on. The drawback of solving a DCSP this way, however, is the exponential growth of the search space, with each step we go deeper in the tree. Additionally, a parsing approach does not account for the error in each observed event, nor does it handle missed observations. To account for these artifacts the parser's rule set would grow exponentially as well.

Existing information retrieval and database query approaches are very powerful tools to filter and sort data. Given the right sorting queries, they are even able to do some rudimentary correlation between data, essentially creating rules of relationships. However, they are not able to express the dynamics of a process that can move through states and have multiple possible observations per state. The following example explains why rule-based approaches have difficulty solving even very simple instances of a DCSP. Recall the symbol x denoting an event not associated with the process $\mathcal{M}_1$. A possible sequence of observed events could be:

$$xxaxcxxbaxxcx$$

A database query or parser/filter rule can be created to only process events a, b, and c. This will filter out all unassociated events x:

$$acbac$$

The next step is to create queries or parser rules that correlate the possible transitions between states as a sequence of observations. Since there are multiple transitions between states, and multiple observable events per state, this group of filter-rules grows quite large (the rules given here associate sequence of observable events to process state transitions):

$$\begin{aligned}
AA &\rightarrow \quad aa \\
AB &\rightarrow \quad aa, ab \\
BB &\rightarrow \quad aa, ab, ba, bb \\
BA &\rightarrow \quad aa, ba \\
BC &\rightarrow \quad ac, bc \\
CC &\rightarrow \quad cc \\
CB &\rightarrow \quad ca, cb \\
CA &\rightarrow \quad ca
\end{aligned}$$

The above set of queries or parser rules are only for single transitions (from one state to the next) in a 1^{st} order Markovian space. To identify sequences of more than two observations in a more general environment, the set of rules must be expanded to detect

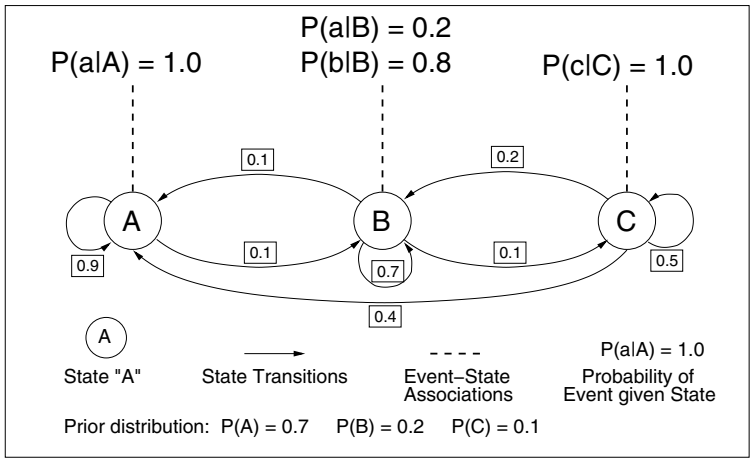


Figure 1.2: A Hidden Markov Model Example, $\mathcal{M}_1'$

all possible occurrences of double transitions (e.g.. state transitions *AAA*, *AAB*, *ABB*, *ABC*, ... etc.). It is clear that the number of queries that needs to be done on a database is growing exponentially. Unless there are no loops in the process graph these sequences are infinite, therefore the number of rules, or database queries are infinite.

An additional difficulty with rule-based approaches arises when the model has probabilities assigned to state transitions and state-event transmissions. Figure 1.2 depicts such a model. In this probabilistic formulation, initial, *a priori* probabilities must also be assigned to states that define the probability of the process starting in a given state. By assigning these probabilities, the process becomes a Hidden Markov Model (HMM) [99].

Given a sequence of observed events and the underlying process model, the DCSP is now defined as the state sequence that *most likely* generated the observed sequence. Viterbi algorithms [43] are well known classical algorithms for solving the "most likely sequence" problem for HMM's. Viterbi-type algorithms use a dynamic programming approach to recursively compute the most likely state sequences at time t_n from the most likely state sequences at time t_{n-1}. The initialization step is determined by the prior probabilities on the states. So, using $s_{1:n}$ to denote a sequence of n hidden states in the process model and $e_{1:n}$ to denote an event sequence of length n, the joint probability

$$P(e_{1:n}, s_{1:n})$$

must be maximized. The point is that standard rule-based approaches cannot handle such iterative state estimation problems (at least not in a scalable manner), and the class of Viterbi-type algorithms cannot deal with multiple different processes. Furthermore, when we are dealing with multiple models, unknown error margins in observations, and possibly missed observations, then any conclusions should have a "confidence" associated with them, indicating how "good the fit" between models and events is. Any

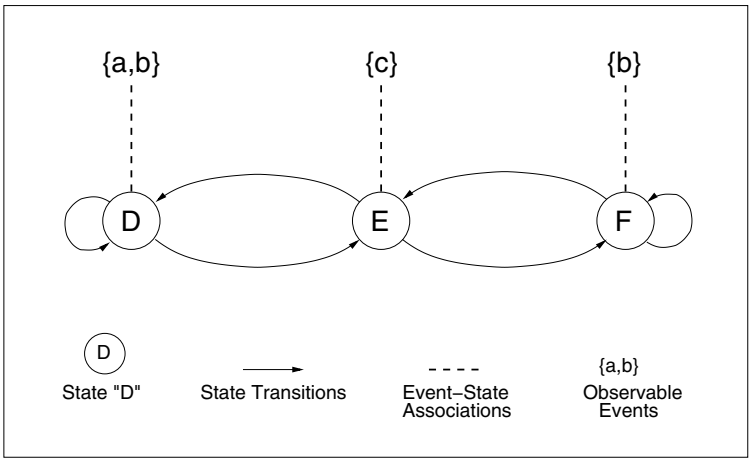


Figure 1.3: Another Process Model, $\mathcal{M}_2$ (compare with Figure 1.1)

rule-based modeling or parsing is unable to do this.

In real-world applications there are problems that further complicate the challenges of solving DCSP's. Missed observations occur when a process is in a state and the observable event is never registered by the sensing framework. Alternatively, the observation is generated, but with a very high probability of error associated with it. The observed process might therefore move to a different state without the Process Query System receiving any observable event indicating a transition might have occurred. In a rule-based approach this would lead to many extra rules accounting for possible changes in state of the dynamic system when no events are registered, and therefore not processed. The complexity of the rule set grows exponentially, together with the cost of maintaining it.

Now consider the challenges that arise when another process model is considered. Figure 1.3 depicts another three state process, using the same notation as process $\mathcal{M}_1$ in Figure 1.1. Note that the states are labeled differently (this is a different process) but that the observable events are collectively the same.

Given an event sequence, say

$$e_{1:9} = aabcbabcc$$

we can now ask the question: "Which instances of which processes $\mathcal{M}_1$ or $\mathcal{M}_2$, produced $e_{1:9}$?" Implying: "Which subsequence of events was produced by which process instance?"

1.2.4 Existing Approaches

The field of tracking (and random signal processing in general) has seen hundreds of very domain-specific algorithms since about 1950, when radar was first used on a large

scale in the second World War (note that the first radar was invented by Sir Robert A. Waston-Watt in 1935). Most of these algorithms worked for one specific instance of the problem domain only, offering little in terms of general usability. The reason for this lack of generality was often the highly specific parameters of the environment in which the algorithms were to be used. Over the years several algorithms were broad enough to stick around and be used on a wider scale. These algorithms are discussed in this section, and compared to the PQS approach. Table 1.1 summarizes the comparison by functionality. The next section will briefly discuss how all of the methods discussed here can be implemented quickly and efficiently using a PQS.

	Formal languages and DFAs	Bayesian / rule-based	HMM, Viterbi	MHT with Kalman Filter	Process Query System
Discrete stochastic processes	Yes	Yes	Yes	No	Yes
Continuous stochastic processes	No	Yes	No	Yes	Yes
Noisy and lossy environments	No	No	Yes	Yes	Yes
Observation disambiguation	No	No	No	Yes	Yes
Multiple process models	No	No	No	No	Yes
Generic model framework	No	No	No	No	Yes

Table 1.1: Table of capabilities of the most prominent and successful methods in the field of random signal processing.

Although Section 1.2.3 illustrates by example why rule-based approaches, databases, regular DFAs, and Viterbi-like algorithms are all insufficient to solve the entire scope of the DCSP, the paragraphs below touch on each of their strengths and weaknesses individually.

Formal languages and state machines

Formal languages and state machines accept, reject, or detect one of many sequences of valid "words" in a string of incoming observations. These methods are more powerful generalizations of rule-based detection and database queries.

A formal language L is usually defined over an input alphabet (of observable events) Σ such that $L \subseteq \Sigma^*$. The language is then defined with a set of production rules over alphabet Σ, allowing the detection of L^+, meaning multiple subsequences of observations that are an instance of the modeled process.

A Deterministic Finite Automata is defined by a quintuple $M = (Q, \Sigma, \delta, s, F)$ where Q are the state symbols to be mapped to the incoming observations. The incoming observations are from alphabet Σ. The mapping must follow the state transitions defined by $\delta : Q \times \Sigma \longrightarrow Q$. The processes that the model detects are therefore deterministic walks through the graph defined by Q and δ, given Σ. Additionally, $s \in Q$ is defined as the *start state*, and $F \subseteq Q$ the group of *final states* [125].

Although multiple instances can be found, only one language can be detected at once (meaning only one process can be detected at once). Difficulty comes when allowing for multiple processes to occur concurrently, thus forcing some initial disambiguation. Since no probabilities are assigned to the various "words" that are detected, all possible solutions would be equally valid. Therefore the number of equally likely possibilities explodes. Intuitively some external method would be needed to decide what to keep and what to toss away. Finally, these methods offer no solutions for missed observations or noise. Generally, an observed event is directly indicative of a state change in a DFA.

To implement either of these methods in a PQS it is assumed that only one instance of one specific process is occurring in the observed environment. All observations are correctly sensed and no observations are lost or altered by noise. The multiple instance, multiple process capability of the PQS must be disabled.

Bayesian and rule-based approaches

Although technically rule-based approaches are unlimited in the number of rules that can be used to achieve PQS functionality, it is straightforward to show that it does not scale as a general method.

Consider creating a set of rules for all the state transitions of each process. Given N models, each with an average of M transitions yields $N \times M$ rules. Now consider that each state of every process can produce up to K different observable events (discrete observation space mapping). Furthermore, consider L rules for each observable to be noisy, or possibly lost. The number of rules is now: $N \times M \times K \times L$, which is only a quantitative measure, the quality of the rules and their order of importance is not even discussed yet.

Some PQS models, however, are served very well by simple rules describing the temporal or spatial causality of process states. These models will use the disambiguation and multiple instance, multiple process capabilities of the PQS, instead of attempting to implement these features directly into the rules.

HMM Viterbi approaches

A Hidden Markov Model is a classical method for modeling processes in a discrete event space where the process states are not directly observable from the incoming events. This means that all states may produce multiple different observables, even the same ones. Model $\mathcal{M}_1'$ in Figure 1.2 is an example of an HMM. Both state transitions, as well as event productions have assigned probabilities. Formally, a Hidden Markov Model consists of a set of states $\omega_n \in \Omega$ with state transitions over time t. These state transitions are time-independent, meaning that the next state only depends on the current state of the process. State transitions probabilities are thus defined as a matrix: $a_{ij} = P(\omega_j(t+1)|\omega_i(t))$. In each state symbols can be produced $e_m \in \Sigma$ with probabilities $b_{jk} = P(e_k(t)|\omega_j(t))$, the matrix of probabilities stating the chance that a given symbol is produced in a given state.

Given a HMM and a sequence of observed events $e_{1...}$ we can now calculate the most likely state sequence of the model based on probability matrices a_{ij} and b_{jk}. Viterbi-class algorithms take the observed event sequence and produce a state sequence that "best" explains the observed events. The specifics of the Viterbi algorithm can be found in many textbooks [43, 99]. Hidden Markov Models and the Viterbi algorithm have been used extensively in speech recognition and data communication.

Although the HMM Viterbi combination is very powerful, it only detects one instance of one process occurring. It has no abilities to disambiguate between observations. Although it is resilient to noise, missed observations must be accounted for in the state transition matrices by adjusting probabilities. To use a PQS for standard HMM Viterbi functionality the multiple instance, multiple process capabilities must be disabled.

MHT, Kalman Filter approaches

Multiple Hypothesis Tracking and the Kalman Filter were initially designed to track aircraft on radar [58, 100]. MHT forms "tracks" of observed events that likely belong together, based on evaluation by the Kalman Filter. In the Kalman Filter the state of a stochastic process is encoded as a n-vector of real numbers, with state x_k at time k. For an airplane the state would include its position and momentum. A matrix F_k is introduced that determines how the next state is calculated out of the current state (in Markovian fashion) in the following way:

$$x_k = F_k x_{k-1} + D_k u_k + w_k \tag{1.1}$$

Where D_k is a matrix modeling the control of the system by changing control vector u_k. (Assuming we know nothing about the pilot's intention, the control term may be omitted.) Finally, w_k is the process noise, assumed to be a normal distribution with mean zero. Observations at time k are defined as: $z_k = M_k x_k + v_k$, where z_k is an m-vector and M_k is an $n \times m$ matrix mapping the observation space to the state space. Finally, v_k models the noise in the observation, considered to be a zero-mean Gaussian distribution. Without going into further detail, the Kalman Filter makes a prediction $\hat{x}_k$ based on F_k and previous state x_{k-1} and the control vector. Simply put, the goal is to minimize the error between the expected observation and the actual observation, where the expected

observation $\hat{z}_k$ is calculated as $\hat{z}_k = M_k\hat{x}_k$. Specifics on the Kalman Filter can be found in the literature, as well as many textbooks on control theory.

MHT has a good ability to disambiguate observations and match them with multiple instances of the process modeled by the Kalman Filter. Although it has multiple instance ability, it does not offer a way to use multiple process models, or process models other than the Kalman Filter. Therefore, classical MHT is not able to disambiguate and identify different objects in the same observation space by their dynamics alone. Most existing solutions are very domain-specific and the lexicon is dominated by radar tracking.

Interestingly enough, many of the problems that can be solved using a PQS with a Kalman Filter model have an optimal solution that reduces to the multi-partite matching problem in graph theory. (This has long been known to be the case for the Kalman Filter.) For instance, consider a rader sweep at time t_0 returning n objects. Now, the next radar sweep, at time t_1, will most likely also return n objects (plus or minus a few). The Kalman Filter can help in matching the objects observed at time t_0 with the objects observed at time t_1, which is the *Bipartite Matching* problem, and is polynomially solvable. However, now consider matching the set of objects returned from a third radar sweep at time t_2, this is the *Tripartite Matching* problem, which was shown, together with any higher order multipartite matching problems, to be NP-complete by R.M. Karp in 1972 [26, 59]. Therefore, PQS offers a heuristic way of approaching the multipartite matching problem, in addition to solving many other problems.

To use a PQS as a MHT Kalman Filter approach, the multiple process model abilities of PQS must be turned off, and the submitted model must be Kalman Filter-based. Many of the multiple instance, multiple model ideas for PQS are based on the MHT Kalman Filter design.

Process Query Systems

The general thread in all the above algorithms is the identification of sequences or groups of observations that are grouped together. All methods provide some sort of score or likelihood for the sequences, yes-or-no for deterministic formal languages and DFAs, actual probabilities for HMMs and the Kalman Filter. Most of the algorithms assume that the groupings are already taken care of, or do not need to be performed, others, like MHT, do the grouping, but do not offer much flexibility in modeling.

Viewed from this perspective, a PQS creates groupings of incoming observations and offers these sequences or groups to the model(s) for this evaluation. The PQS takes care of all these groupings, called tracks. Models therefore may contain Bayesian rules, state machines parsing formal languages, the Kalman filter, the Viterbi algorithm with one or more HMMs, or may simply be a whole new type of scoring algorithm.

The models are allowed to keep state with groups of observations, such that they will not have to recalculate the scores each time they are invoked. There is a guarantee that no more than one new observation will ever be added before the model is called again. This way a PQS creates an abstracted level where the application programmer only needs to work on implementing the proper models. All observation handling, pairing, and pruning is taken care of by the PQS.

1.3 Solution Roadmap

This section outlines all the areas of development that need to be addressed to fully complete the PQS concept. It offers a road map of tasks and states that have been completed and implemented by this thesis. Some steps are handled in depth by this thesis, others are only touched upon lightly.

- *Problem Definition.* Description of the problem domain, existing algorithms and approaches, and a formal problem definition. This is given in Chapter 1.

- *Generic multi-datastream, multi-model engine.* This is the core of a Process Query System. The concept of a generic engine with the described functionality involves several sub-steps:

  1. *Multiple Datastream.* The ability to hook into multiple arbitrary datastreams easily and quickly. This requires a flexible interface in the PQS to define the format and origin of the incoming data.
  2. *Event grouping.* Basically the creation of tracks and hypotheses. This has to be internally consistent and complete. No hypothesis may ever have more than one copy of each observation, and all possible pairings into tracks should be allowed.
  3. *Model scoring.* The grouped events (called tracks) will be offered to all the models for evaluation. In a generic fashion, the models will be plug-ins that take a track of observations and return a score indicating how well the observations "fit together". This may involve predictions, state sequences, or simply a binary yes-or-no answer.
  4. *Hypothesis pruning.* Based on the scores assigned by the models, the majority of hypotheses should be pruned. It is inevitable that most event groupings will score very low, as they do not belong together. Pruning keeps PQS scalable.

  The above topics are fundamental to the functionality of a PQS and are fully explored in this thesis, Chapter 2.

- *Advanced Hypothesis Pruning.* Although a PQS functions well with several fundamental pruning methods (those described in this thesis), there are more advanced pruning ideas that justify investigation. For instance, it would be very beneficial to compare, in a generic, application-independent way, how different two hypotheses are. Two very similar hypotheses are obviously both going to score quite high and survive pruning, whereas a slightly less optimal, but potentially very promising hypothesis will be cut. One way of tackling this problem is by comparing the parental history of each hypothesis, and then to disadvantage hypotheses that are very close together in the family tree. A thorough study remains to be done on the merits of such an approach.

- *Modeling interface.* The language in which models are interpreted is called PQML: the Process Query Modeling Language. It is explored with several examples in Chapter 2. However, this language is the lowest level modeling interface possible. Specifically, all higher level models such as HMMs, DFAs, Rule-based, Kalman Filter, etc. must currently be written directly in PQML. Although PQML is powerful and described in-depth in this thesis, writing higher level models directly in PQML is not always desirable. One or more higher level modeling interfaces still need to be designed.

- *Higher level modeling interfaces.* A higher level modeling language, or multiple different higher level modeling interfaces would be very valuable additions to PQS. Such languages would require compilers that can compile to PQML, or, even more efficiently, directly to machine code. At present no such compiler exists. Additionally, it would be very beneficial to allow models an extra interface through which they can decide whether or not an observation may be a likely match with a track. Such a function would be boolean and could significantly reduce the number of hypotheses that the PQS generates at each step. Although such an interface would not lead to better PQS tracking accuracy, it would make the core algorithms much more efficient. This interface has not yet been designed or investigated.

- *Performance evaluation.* PQS performance has two different aspects: (1) tracking accuracy, and (2) computational complexity, or efficiency. Tracking accuracy pertains to the results produced by the PQS and the supplied models. The better the results, the higher the accuracy. Computational complexity can be measured in memory and compute cycles per observation. It has a direct relationship to the amount of work that needs to be done for each observed event. Both measures are closely related, however. Better models often lead to less computational complexity and higher accuracy, while a noisy environment tends to increase computational complexity and negatively impact tracking accuracy. Although both factors are highly environment-dependent, the topic of performance is discussed throughout this thesis.

- *Auto-configuration.* The initial implementations of PQS show that there are many parameters that need tuning for each specific application. These parameters include the number of hypotheses, length of tracks, hypothesis scoring methods, etc. Based on experience so far, it is often possible to quickly tune the parameters for a specific application; however, ideally the PQS would measure its internal performance and adjust these parameters dynamically. The performance sections give some idea on how these parameters could be automatically configured, although a thorough study remains to be done.

- *Applications.* Even though PQS is a generic foundation for many different applications, by itself it can only be regarded as an academic framework. To demonstrate the usefulness and applicability it must be used to implement multiple applications. During the last two years, PQS has been used in many different ways, and this thesis presents several powerful applications in Chapters 3, 4, and 5.

The above list is by no means exhaustive and many smaller topics remain to be addressed. For instance, although this thesis describes a way to compute the scoring correlation coefficient of two models, effectively computing how similair they are, it tells us nothing about the intended purpose of the models. This comparison is not necessarily a scientific process, moveover, comparing models is generally a matter of semantics. Therefore, what is *intended* by the model builder is more relevant than what the actual model code looks like. What the scoring correlation coefficient tells us, however, is how similar the scoring behavior of two models is. Although this has little to do with how accurate and efficient each model is, we can learn by submitting a broad range of tracks to both models whether their scores show any relationship (through straightforward covariance calculation). For example, if two models are submitted, one describing the dynamics of a truck, the other describing the dynamics of a car, it is expected that both will return similar scores. Only when it is very clear that we are dealing with a car, will the car model differ much in its score from the truck model, and vice versa. It is evident that some of these questions are very difficult to answer, and this thesis only offers considerations for them. More about comparing models can be found in Section 2.1.2.

1.4 Performance measurement

When such a complex system as a PQS is implemented, it is necessary to have strict performance measures in place to evaluate if a PQS-based solution is working as expected. Performance, however, is difficult to define precisely because, in practical operation, a PQS is expected to discard unlikely hypotheses and thus lose information. This means that there is a chance that the actual correct solution may not be found, and a slightly less optimal solution may be returned instead.

There is not much that can be said about the performance of Process Query Systems before the algorithms are explicitly discussed (in Chapter 2), although one distinction can be made now. Performance can be defined on two areas: (1) the accuracy of the outcome, and (2) the computing resources needed to reach that outcome (also referred to as efficiency). Intuitively *more computing resources lead to higher accuracy*, but that is not necessarily true. Specifically, both accuracy as well as efficiency are highly dependent on the application that the PQS is used for, although general trends can be identified. Consider, for instance, sampling frequency; if the sampling frequency increases, the total amount of work per second will increase, however, the amount of information about the environment will increase as well, thus making the problem easier to solve. This will lead to higher accuracy, and probably higher efficiency as well.

At the end of every Chapter there is a discussion regarding both the accuracy of the application, as well as its efficiency. Both measures are especially important when scalability and applicability are concerned.

1.5 Thesis outline

During the research stages of this thesis, many pieces of software were developed, including two general purpose Process Query Systems. The first, experimental PQS is named TRAFEN, which stands for TRacking And Fusion ENgine. TRAFEN is an implementation of a general purpose PQS. All applications described in this thesis were based on TRAFEN (by simply submitting the domain-specific models). In Chapter 3 a system is described called DIB:S, the Dartmouth ICMP Bcc: System. This piece of software was written before TRAFEN and was made to process ICMP Destination Unreachable messages in the same way a general purpose PQS processes observations. Later on, DIB:S was combined with TRAFEN to form a powerful Internet worm detection system. DIB:S is the first stage in this worm detection system, handing its processing results to TRAFEN, which then draws conclusions. The remainder of this thesis is organized as follows:

Chapter 2. This chapter describes the internals of Process Query Systems, how models can be defined and compared, and the pruning and hypothesis generation algorithms. Examples are used to show the reader the most relevant data structures and methods to clarify the inner workings of TRAFEN, an implementation of a PQS. It continues by describing PQML, the Process Query Modeling Language, and giving an annotated example, where a PQML model is used to configure a PQS to do deterministic sorting. The chapter closes by analyzing PQS accuracy and efficiency using an environment where balls bounce around inside a 2 dimensional box. Parts of this chapter have been published in [9] and [31].

Chapter 3. The Internet worm detection and tracking system DIB:S/TRAFEN is described in-depth in this chapter. The DIB:S system aggregates ICMP Destination Unreachable messages from routers across the Internet to catch any hosts showing aggressive scanning behavior. TRAFEN is used to correlate the output from DIB:S, creating a system that is capable of detecting worms within minutes, sometimes even seconds of their initial release. The internal architecture of DIB:S was an important motivator for the first designs of the Process Query System concept. Parts of this chapter have been published in [6] [8] [11] [12] [47] and [69].

Chapter 4. The PQS-Net system described in this chapter is a powerful example of the general applicability of a PQS. The system is capable of detecting and prioritizing security-relevant processes in large, enterprise-class computer networks. The chapter describes many different sensors and models that were used with TRAFEN to create the PQS-Net system. A part of this chapter has been published in [10].

Chapter 5. This chapter describes three very different, and very succesful applications based on the TRAFEN PQS implementation. The first application is a detector for *timing covert channels*, which are created by modulating the delay times between packets of benign traffic to encode data. This way an attacker can exfiltrate data without

ever needing to create traffic on the network; instead, the timing of exising traffic is perturbed. Then we describe the use of a PQS to track fish swimming in a tank. The fish are filmed and X,Y-coordinates are generated, which are then fed into TRAFEN. Using a Kalman Filter-like model, the fish are tracked as they move around obstacles. Finally, the chapter describes an autonomic computing server monitoring system. This system monitors large networks of servers for potential failures or degredation of service, and stops or restarts services when failures or intrusions are detected. Ideas from this chapter have been published in [102].

Appendix A. Finally, this appendix contains the PQML specification document. This document precisely defines how a PQS should handle a PQML model file. It is the lowest possible level for defining implementation-independent PQS models.

Chapter 2

Process Query Systems

In this chapter the specifics of implementing a Process Query System are discussed by describing the internals of TRAFEN, an implementation of a general purpose PQS. The algorithms are shown in detail and an example is developed. The chapter ends with an in-depth analysis of the performance of a PQS. A PQS is used in an experimental setting to verify the analytics of performance and complexity.

2.1 Fundamentals of Process Query Systems

Many DCSP and related problems are very hard to solve exactly in finite time, especially when lost or incomplete observations are considered. In practical applications it is often required to present partial solutions, or hypotheses, while observations are coming in. The partial solution must be as accurate as possible given the models, such that practical decisions can be made. The Process Query System paradigm is designed to continuously present these hypotheses explaining a constantly changing observed environment. PQS is a processing kernel that abstracts away the management of observations (observable events), the overhead of dealing with missed detections, and lets the user focus on building models. Building models, then, is the main task of the application programmer using PQS. It is the place where all expert knowledge goes into the system. Models can be written from first principles, mathematical models, or can be learned using a feedback loop. An in-depth discussion of model building follows later on in this Chapter.

2.1.1 System overview

The creation of hypotheses out of the received observations handled internally in the PQS and the most feasible explanation of the observed environment is presented to the user. Figure 2.1 gives a high-level overview of the components of a Process Query System core.

As the figure suggests, the PQS core contains several important components. These components are designed such that the user of a PQS only has to focus on the creation of models and connecting the sensors as input. Since models, like the sensors, are externally supplied, they are not part of the PQS core components. Specific functions of each of the core PQS components are:

- *Input Observation Handling.* This component is a generic and flexible subscriber that can take input from sensors in arbitrary format. The specific implementation of a PQS may dictate how this component functions, but it is very important that this part is tightly integrated with the specification of the models (the Process Query Modeling Language, PQML, defines such an intergration, as will be described later). Specifically, the sensor subscriber must be able to take input in many different formats, handle sensor specific errors, and subscribe to many sensors at once. Additionally, this component must allow for the mapping of variables in the models to fields in individual observations. Finally, most sensors have a known error margin, which is encoded into the observation as a *fidelity*, indicating

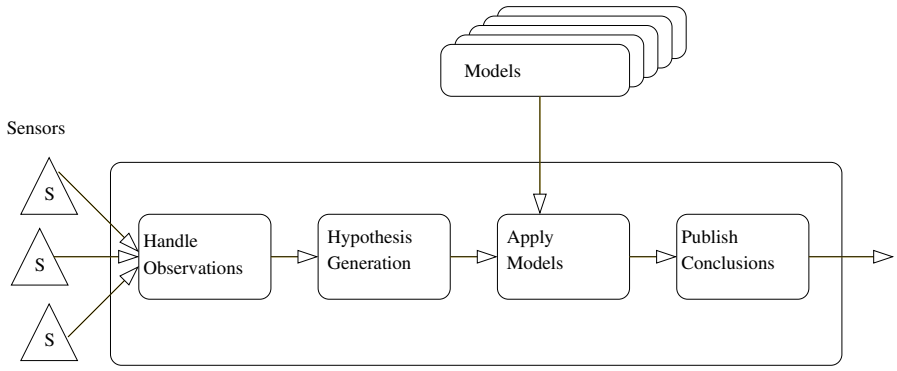


Figure 2.1: High level overview of the components of a PQS system. The models and the sensors are the only external components that the user needs to focus on. The sensors determine the input to the PQS, the models determine the behavior.

how much this observation can be trusted. We will refer to observations with the symbol O_t, indicating that this observation was generated from event e_t.

- *Hypothesis Generation.* This part takes all the incoming observations and groups them together in "tracks", refered to as $\mathcal{T}$. When multiple hypotheses are generated, they must all be internally consistent, meaning that no hypothesis may have multiple copies of one observation. Each hypothesis, refered to as $\mathcal{H}$ represents an *alternative view of the observed environment*. The most general (and so far successful) approach to hypothesis generation has been to simply generate all possible combinations of new observations in tracks in hypotheses, and have an extensive pruning step after model scoring is complete. In this way only the most likely and consistent hypotheses remain in the system.

- *Model Scoring.* The next step takes the set of hypotheses and applies the models. The models "score" how well each hypothesis fits the set of process descriptions. This means that, given the process descriptions in the models, how well do the hypotheses explain the observed events? In this step it is important to realize that the models access the actual observations. Variable names in models must therefore correspond to fields in the observations coming from the sensors (for instance: if a sensor uses a *src_ip* field in the observation, then the model must be able to access that field by the same name).

- *Conclusion Publication.* Based on the scores assigned by the models to the hypotheses, the last step decides which hypotheses to output to the user. Intuitively the hypothesis that is most internally consistent will score highest and thus should be published. The fidelity (usually $1 - error$) is amortized over all observations to generate a confidence, indicating how trustworthy the conclusion is.

Additionally, an external component not necessarily included in the PQS is a set of performance metrics to evaluate how well the PQS is performing given the set of models and input sensors. Since performance evaluation is usually application-specific, it is not part of the PQS core. Possible problems include the lack of sufficient input observations from the sensors, and improper scoring by the models. How to measure the performance is up to the requirements of the specific application.

2.1.2 Model overview

Thanks to the *Input Observation Handler* the format and protocol of the sensor observations can be arbitrary. The models, however, are a little bit more involved, and are essentially the plug-ins that apply a PQS to a specific domain. The PQS takes care of making many groupings of observations into different tracks, and offers these tracks for evaluation to the models. The models, then, return a score indicating how much the track is evidence of the modeled process. At the lowest level, the API that the model should offer the PQS is:

```
double Halflife;
void *State;

double Score (Track *t)
{
    // Evaluate Track t
    Set(t->State);
    return(score);
}

conclusion *Conclude (Track *t)
{
    Get(t->State);
    // Build a conclusion
    return(conclusion);
}
```

Where *Halflife* describes the rate of decay of previously assigned scores (more about this in Section 2.2.7), **State* is the current state of the process in a track, and should be set by the *Score* function. Since **State* is specific to each track, a unique copy needs to be made for each track. The *Score* function, while calculating how much a given track is evidence of the modeled process, also determines the current state of the process and stores that with the track. Every time a new observation is added, the *Score* function must be called

again to re-evaluate. Finally, when a track is published the **Conclude* function is called which generates a datastructure from **State* that is a conclusion for the track. Note that by design, scoring and publishing of conclusions are independent. Arguably a third function can be added:

```
boolean mayMatch(Track *t, Observation *o)
{
    if (fitsWithState(t,o)
        return(true);
    else
        return(false);
}
```

indicating whether a new observation would fit with the existing track at all, and should even be tried with the *Score* function. The *mayMatch* function would thus have to be much more lightweight than the *Score* function. Such an addition could potentially save a lot of processing time in very busy environments where there are a lot of processes happening at once. Models can come in many different forms:

- *State Change over Time.* This is a very intuitive form of model because it is very applicable to kinematic tracking (the tracking of objects through 2 or 3 dimensional space, over time.) The closer the observation sequence follows the $\frac{\Delta x}{\Delta t}$ defined by the model, the higher the score. More sophisticated models can have varying Δx, or even learn an objects specific variations for Δx.

- *Noisy State Change.* This is actually the same as the above, however, the model needs to account for noise (inaccuracy) in the observation stream. The Kalman Filter is very well suited for this type of model, defining the process state at time k as $x_k = F_k x_{k-1} + D_k u_k + w_k$, a function of the process dynamics modeled in matrix F_k, and noise modeled in vector w_k. Knowledge of input controls to the process can be modeled with D_k and input vector u_k.

- *Discrete State Change.* This is a subclass of the first type of model where Δx happens in discrete, predefined steps. The sequence of observations can be directly mapped to a sequence of valid state transitions. This type of model is often defined most easily with as a Markov chain (with probabilities) [99], or a Petri Net (without probabilities) [125].

- *Hidden Discrete State Change.* This is a special case of the third (Discrete State Change) models where the observations cannot be mapped to discrete states directly. Models of this type are best represented using Hidden Markov Models, where, besides likelihoods for each state transition, there is also a likelihood associated with all possible observables for every possible state.

- *Rule-based Models.* Some models are best represented as simple **if-then-else** clauses; for example, when a sequence of observations indicates a single state change, or when a sequence of observations raises a flag. Often simple rule-based models can perturb the *previously assigned score*, that they themselves generated, leading to very complex model behavior using only very simple rules.

- *Formal Language Based Models.* Most models that are expressed as formal languages can be also be expressed as one of the other model classes discussed above. For example, given is formal language L over alphabet Σ such that $L \subseteq \Sigma^*$. When observations are mapped to alphabet Σ and the model recognizes L^+, then the model can be implemented as a Petri Net, or a Finite Automata as well. Some models are simply better expressed as a formal language, whereas others are better expressed as an automata.

- *Deterministic Finite Automata.* This class of models is very broad and has significant overlaps with formal language-based models, Markov and Hidden Markov models, and Petri Nets. The DFA is defined as a quintuple $M = (Q, \Sigma, \delta, s, F)$ where Q are the state symbols to be mapped to the incoming observations. The incoming observations are from alphabet Σ. The mapping must follow the state transitions defined by $\delta : Q \times \Sigma \longrightarrow Q$. The processes that the model detects are therefore deterministic walks through the graph defined by Q and δ, given Σ. Additionally, $s \in Q$ is defined as the *start state*, and $F \subseteq Q$ the group of *final states* [125].

Finally, it is often necessary to compare models to determine why PQS performance is lower than expected. Comparison of different models depends once again on the problem domain and the specification of the particular models. One way of comparing two models $\mathcal{M}_1$ and $\mathcal{M}_2$, with scoring functions $S_k^{\mathcal{M}_1}$ and $S_k^{\mathcal{M}_2}$ respectively at time time k, is to experimentally determine their scoring covariance.

$$cov(\mathcal{M}_1, \mathcal{M}_2) = E((\mathcal{M}_1 - \mu)(\mathcal{M}_2 - \nu)) \tag{2.1}$$

where $E(\mathcal{M}_1) = \mu$ and $E(\mathcal{M}_2) = \nu$, the expected scores returned by the models for all possible tracks. Since it is impossible to calculate the expected score for all possible tracks (infinite), we must settle for an approximation. Given a sufficiently large number K tracks, then for all tracks $\mathcal{T}_k$ for $k \in K$ we have:

$$E(\mathcal{M}) = \sum_{k=0}^{K} \frac{1}{K} S_k^{\mathcal{M}}(\mathcal{T}_k) \tag{2.2}$$

which is the average expected score that model $\mathcal{M}$ would return were we to pick a track randomly from our collection. Therefore the scoring covariance between two models can be computed with:

$$cov(\mathcal{M}_1, \mathcal{M}_2) = \left(\sum_{k=0}^{K} \frac{1}{K} S_k^{\mathcal{M}_1}(\mathcal{T}_k) S_k^{\mathcal{M}_2}(\mathcal{T}_k) \right) - \mu\nu \tag{2.3}$$

If the models are fully independent we would expect their covariance to be zero, keeping in mind that only a finite number of tracks can be given to the models to compute this number. However, since models are not necessarily bound to scores between 0 and 1, it would make sense to normalize the average expected scores such that other values for the covariance also make sense. We give a normalized version of the covariance using a correlation coefficient:

$$\rho_{\mathcal{M}_1,\mathcal{M}_2} = \frac{cov(\mathcal{M}_1,\mathcal{M}_2)}{\sqrt{var(\mathcal{M}_1)}\sqrt{var(\mathcal{M}_2)}} \tag{2.4}$$

where the variance $var(\mathcal{M})$ is the average quadratic deviance from the average scores assigned by $\mathcal{M}$. This variance number can be computed as follows:

$$var(\mathcal{M}) = E(\mathcal{M}^2) - (E(\mathcal{M})^2) \tag{2.5}$$

expanded as:

$$var(\mathcal{M}) = \sum_{k=0}^{K} \frac{1}{K}(S_k^{\mathcal{M}}(\mathcal{T}_k))^2 - (\sum_{k=0}^{K} \frac{1}{K} S_k^{\mathcal{M}}(\mathcal{T}_k))^2 \tag{2.6}$$

The correlation coefficient is normalized and bounded: $-1 \leq \rho_{\mathcal{M}_1,\mathcal{M}_2} \leq 1$, where a value of 0 means that both models assign their scores fully independently, whereas a value closer to -1, or closer 1, means that the models have an increasing degree of correlation in their score assignments. Specifically, if $\rho = 0$, then the score assigned by model $\mathcal{M}_1$ tells us nothing about the score that model $\mathcal{M}_2$ will give to that same track. If ρ gets close to -1, then we know that a "high" score by model $\mathcal{M}_1$ will likely mean that model $\mathcal{M}_2$ will give a relatively "low" score to the same track, and vice versa. Likewise, if ρ is positive, then it may be expected that a "high" score assigned by model $\mathcal{M}_1$ means that model $\mathcal{M}_2$ will also assign a "high" score for that same track.

Interestingly enough, these quantities can be computed on the fly by the PQS when two or more models are submitted. Then, when one model seems to rarely score tracks higher than other models, the correlation coefficient can tell us if there is another model that assignes similar scores to the same tracks and is most likely dominating the weaker model. Finally, poor performance can often be linked to a lack of sensors, or a poor sampling rate. Poor model design can often be linked to low track scores, or many short tracks, while the sensor data is clearly good. It must be said, however, that it is very difficult to generically determine if the sensors or the models are at fault.

2.2 An implementation: TRAFEN

Conceptually a Process Query System takes arbitrary observations and process models as input, and produces output that shows whether evidence of the process models is present in the observation streams. This section describes our implementation of a PQS, called TRAFEN, the TRAcking and Fusion ENngine. TRAFEN is based on a Multiple Hypothesis Tracking approach, relying on some fundamental concepts from Evolutionary

Computing. As a one-sentence overview, we could describe TRAFEN as a multiple hypothesis core, where hypotheses are scored for fitness by multiple process models, and the hypotheses are built from the incoming observations. Note that TRAFEN is the PQS implementation that is used for all applications in this thesis.

This section starts with a thorough description of the hypothesis management scheme: how observations are organized into tracks, how tracks form hypotheses, and how hypotheses form hypothesis-sets. Next are the process evaluation methods and the pruning system. Then we discuss the specifics on the input, output and internal state of the system. We will end with a discussion of the Process Query Modeling Language PQML, an assembly-inspired language that allows fast and easy specification of observations, models, and conclusions.

2.2.1 Observations, Tracks, Hypotheses, and Hypothesis-sets.

The hierarchy of data in TRAFEN is organised in four levels: Observations (O); the data coming directly from the sensors, Tracks ($\mathcal{T}$); sequences of related observations given the models, Hypotheses ($\mathcal{H}$); sequences of tracks, representing a whole (albeit not necessarily correct) view of the system, Hypothesis-sets ($\mathcal{W}$); sequences of hypotheses, holding a range of views of the system.

Each hypothesis represents a complete view of the world. Among hypotheses in a hypothesis-set there will be commonalities, as well as differences. No hypothesis claims to be a complete and accurate view of the world, however, over time some hypotheses may be more prevalant and accurate than others. In that sense, as more information comes in, it might become evident that a "less likely" hypothesis provides a more accurate view of the world, therefore becoming "more likely". What "likely" means is dependent on the process descriptions in the models. It is therefore important that multiple different hypotheses are kept around, because the views on the world will change over time, as more evidence comes in.

A hypothesis is a collection of tracks, where each track is a collection of observations that "fit together". This means that together they provide evidence of a process happening. A track therefore could be seen as the collection of observations that supports the occurrence of a given process. If a process is happening multiple times in the observed world, there will be multiple tracks, each with a set of observations supporting one particular instance of the process. Assume, for example, that TRAFEN has one process model describing the parabolic arc that an object follows when thrown upwards. When two balls are thrown up in the air, observations will be coming in, indicating the position for two balls. Unless the balls are thrown at exactly the same time, with exactly the same force, at exactly the same position, it will be very unlikely that the observations can be confused for a single ball. Therefore, under the given process model, two tracks, one for each ball, will be much more likely than one big track with all the observations. The single big track would make it look like the ball made jumps in time or space and would therefore not get scored very high by the process model.

To get a better understanding of the logic we will now dicuss the algorithm that drives

the system internally. The similarities with Multiple Hypothesis Tracking [100] should be apparent to the reader.

2.2.2 Newly incoming Observations

In short, every new observation needs to be evaluated against every track, by every model in every hypothesis, meaning that all combinations must be tried. So, the observation needs to be added to every track in the system, and then evaluated against all the models. However, because every hypothesis represents a whole view of the world, it would be inconsistent to simply add the new observation to every track. This is because an observation "occurs" only once in the observed world. This means that a hypothesis needs to be cloned for every track that the observation is "tried with".

Cloning of a hypothesis simply means that an exact copy is made of the hypothesis, and the new observation is added to a different track for every cloned copy. All these modified clones are added to the hypothesis-set $\mathcal{W}$. In this way there is never more than one copy of an observation present in each hypothesis, while it can still be tried with every track in a hypothesis. This cloning of hypotheses brings with it an exponential explosion in the number of hypotheses that are present in the system, and some type of hypothesis management needs to be done. Later we will talk more about pruning methods.

This method of hypothesis generation is brute force and creates an entirely new hypothesis for every track in each hypothesis, for every new incoming observation. Since there is no general way to control this exponential explosion, several important other steps must be taken as well:

- *Pruning.* Since there will be only very few combinations of the new observation with the existing tracks that make any sense, all other combinations must be pruned away immediately, preferably before the next observation arrives. Since the system relies on the model scoring to determine what combinations are reasonable, all generated hypotheses and tracks must be evaluated by the models first. After this evaluation cycle only several of the very best hypotheses should be saved. Pruning therefore happens every time right after evaluation by the models, which, in turn, is done for each observation that arrives. (And in some cases multiple times between two observations.)

- *Copy-On-Write.* When a hypothesis is cloned, all its tracks are cloned with it. Since, in general, only one track of the cloned hypothesis will be modified, it is unnecessary to actually copy all the other tracks. In any software implementation it is therefore important that actual copies are only made when the track is modified. Likewise, when models evaluate tracks the software implementation should ensure that each track is evaluated only once.

Additionally, the new observation might be evidence of a whole new event (or process) happening. This means that a track must be created with merely this new obser-

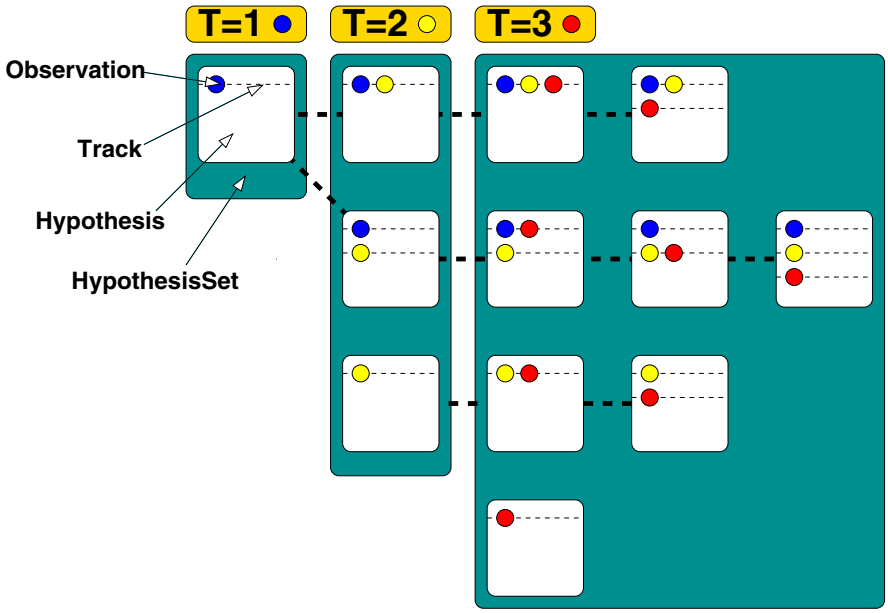


Figure 2.2: Arrival of 3 new observations in an empty hypothesis-set. The outer box (green) is the hypothesis-set containing the hypotheses (white boxes). There is only 1 hypothesis-set per time step. The hypotheses contain tracks (dotted lines) with the observations (the blue, yellow, and red dots).

vation, and this track is then added to a copy of the original hypothesis. To illustrate the cloning of hypotheses and construction of tracks as new observations arrive, consider Figure 2.2. Initially the hypothesis-set is empty. The hypotheses in the hypothesis-set change with the arrival of observations. The hypothesis-set is indicated with the large dark green boxes. At time $T = 1$ the first observation comes in. In the hypothesis-set a new hypothesis is created. In this hypothesis a new track is created, which holds one observation: blue. Each hypothesis is indicated with a white box, and tracks are the small dotted lines in the hypotheses that hold the observations together. The thick dotted lines, from hypothesis-set to hypothesis-set indicate how hypotheses evolve over time.

So, at time $T = 1$, after the blue observation came in, the hypothesis-set contains one hypothesis, which contains one track, which consists of one observation: the blue one. At time $T = 2$ another observation comes in: the yellow one. The only hypothesis in the hypothesis-set is cloned once. In the first copy the yellow observation is added to the existing track. In the second copy the yellow observation is put in a new track, all by itself. Finally, a third, all new, hypothesis is created that holds one track, which consists of only the yellow observation. Now the hypothesis-set contains three hypotheses, and four tracks. Note how none of the hypotheses contains more than one copy of an observation.

At time $T = 3$ the third observation comes in: the red one. The first hypothesis (with

one track: blue-yellow) is cloned, the first copy gets red added to the existing track, and in the second copy the existing track is not modified and a new track is added, consisting only of the red observation. The second hypothesis (with two tracks: blue, and yellow) is cloned twice. The first copy has red added to the blue track, the second copy has red added to the yellow track, and the third copy gets a whole new track with red all by itself. And it goes analogously from here for the third hypothesis from $T = 2$. Finally a whole new hypothesis is added with one track, consisting of only the red observation all by itself.

Note that in Figure 2.2 a new hypothesis is generated at each step ($T = 1$, $T = 2$, and $T = 3$) that contains one track with only the new observation. This is a strategy to get tracks and hypotheses started in a condition where the hypothesis-set is empty. The hypothesis-set is empty right after starting the algorithm, and at other times when pruning has eliminated all tracks and all hypotheses. In most other cases (when there are other tracks and hypotheses present) this new *singleton* hypothesis is usually of very little consequence, for it tends to get pruned right away. Experience has demonstrated that disabling creation of this singleton hypothesis does not effect the tracking abilities of the algorithm.

Technically, to preserve symmetry, it would be justified to also maintain a copy of each of the original hypotheses. This would mean that at time $T = 2$ the single hypothesis from $T = 1$ is copied and maintained unmodified. The assumption would be that the newly received observation is considered noise by the geometry of the algorithm, instead of by model scoring. However, intuitively it can be guessed that this may get the algorithm stuck in a local maximum. Consider a hypothesis that is scored very high by the models, meaning that this hypothesis is very favorable. A newly incoming observation is therefore likely to create a less favorable set of hypotheses. This would mean that all hypotheses with this new observation present will be pruned, leaving only the original umodified one. This local maximum has indeed been observed repeatedly in experimentation, thus getting the PQS "stuck". To avoid this problem the original hypothesis is never copied verbatim into the next hypothesis set.

The nature of this algorithm leads to exponential explosion of hypotheses and tracks over a very short amount of time. When we keep in mind that tracks are sequences of related observations, given a certain model, then we can conclude that most tracks and hypotheses that are formed in the outlined algorithm are of very little value. To put it another way, a Process Query System tries to detect a process in an incoming observation stream. That means that the process model must evaluate all possible orderings of observations (the tracks) and decide if they are evidence of the process occurring. All other combinations are of no (or lesser) value, and may be discarded.

2.2.3 Track Scoring with Models

To decide which tracks and hypotheses can be thrown away (pruned), the models need to evaluate the tracks and store the updated process state. When a model finds a track to be possible evidence of the process that it represents, it gives the track a "high score". When

a model finds a track to be poor evidence of the process that it models, then it gives the track a very "low score". This score measures how *likely* it is that observations in the track were formed due to the modeled process occurring. From now on we will refer to this "score" as the "likelihood" (that the track is evidence of a modeled process).

For example, assume process model $\mathcal{M}_3$ describes a process in which there are an arbitrary number of yellow observations, and then a red observation: $[Y^*R]$. (This could be modeling the failure of a network device, a traffic jam, a factory line failing, etc...) Model $\mathcal{M}_3$ will therefore give a high likelihood to any track that has yellow observations followed by red. If there is just a red observation, the likelihood will be relatively high as well. Any other combination of observations in a track will be scored with a low (or zero) likelihood by model $\mathcal{M}_3$.

For comparison, assume a second process model $\mathcal{M}_4$ describing a process in which a blue observation is always followed by a yellow observation. Therefore, this model will score a lone blue observation fairly high, in anticipation of what may come next, and it will score any track with blue and yellow very high, to indicate the detected process. To clarify this example consider Figure 2.3. After the cloning is done, and the new observation has been added to the appropriate tracks the two models are asked to evaluate each track. The golden boxes give two scores, the first one from $\mathcal{M}_3$ and the second one from $\mathcal{M}_4$. So, the first hypothesis of the second hypothesis-set, where $T = 2$, has one track consisting of one blue observation and one yellow observation. Model $\mathcal{M}_3$ scores this track with a likelihood of 0.2 to indicate the perceived match of the yellow observation at the end of this track. However, model $\mathcal{M}_4$ scores this track with a likelihood of 0.7, because it is a very good match with the yellow observation following the blue observation. This example shows how models may overlap in their scoring. In fact, if we look at the first hypothesis in the third hypothesis-set, at $T = 3$, we see that it has one track with three observations. Both models scored that track very high, because the track is a good match for both models. This means that the track could be considered evidence of both processes occurring.

2.2.4 Pruning and Hypothesis Control

Given the scores that are assigned by the models, it is very straightforward to decide which tracks should be discarded. Because all possible combinations are made, of tracks and arriving observations, there will be many tracks that won't score very high (see Figure 2.3). A track which scores low for all models is a candidate for removal. It means that the combination of observations in that track do not represent any of the processes that the models describe. On the other hand, if a track is scored high by any model it should be kept. Even if there is only one model that scores a track high, it means that the track is likely to have been produced by the process that the model represents. In fact, the modeled processes should be very different from each other and therefore most tracks will get a high score from *only one* of the models. To conclude, the ultimate "total" score of a track should be the highest score that *ANY* model assigned to that track.

To decide which hypotheses make the cut we need to assign a score to each hypothesis as well. Intuitively the hypothesis score should reflect how well the hypothesis as a

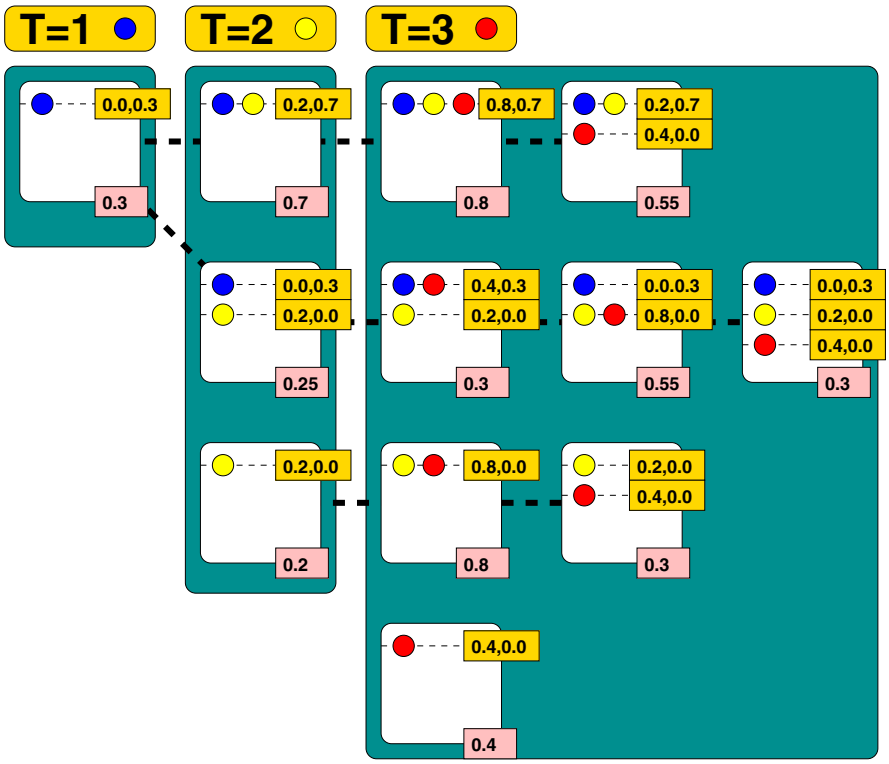


Figure 2.3: Scoring given models $\mathcal{M}_3$ and $\mathcal{M}_4$.

whole explains the observed sequence of observations. Considering that the incoming observations are produced by processes happening in the observed world, the hypothesis score should reflect how well its tracks are scored by the models. After all, the models model the processes occurring in the observed world. If no processes are occurring, and the observations can therefore not be properly grouped in tracks, the score of all the tracks will be low for all the models. Then the hypothesis score will be low also, which is correct. If one or more processes are occurring in the observed world, tracks will be scored high by the models, and thus the hypothesis score must be high to reflect a good explanation of the sequence of observations. This is done by taking the highest track scores of all the tracks in the hypothesis, and averaging them. This can be done straight, or in a weighted manner, where the size of the track is made to count as well. The example shows the hypothesis score (in the pink boxes) as the average of the highest scores of all the tracks in that hypothesis.

Note that the example in no way reflects the definitive way in which a PQS generates an overall score for the hypothesis. The score of the hypothesis must represent how well the collection of its tracks represents processes in the observed world, given the models.

If, for example, an engineer is trying to build a radar system capable of tracking airplanes near a bird nesting area, relatively speaking, there will be few planes and a lot of noise (the birds). So, there will be many observations with many different radar footprints, most of which are probably merely interference. The process model would describe the exact properties of airplanes such as the speed and expected radar footprint. The hypotheses will most likely contain several large tracks with the observations that are evidence of aircraft, and many small tracks with all observations that can be considered noise. The engineer will therefore only be interested in the long tracks and not care at all about the noise tracks. He might therefore argue that the score of the hypothesis should be the score of the highest scoring track.

Another way of dealing with this specific case would be to insert two models into the PQS: one that describes airplanes, and one that describes *everything but* airplanes. The second model could be most easily constructed by returning a score: 1 - score of the airplane model, and so effectively modeling all non-airplane noise. In this way the hypothesis score could easily be computed as the average of the highest track scores, and everything would work fine. For convenience there are several different methods of creating a score for the hypothesis, given the highest score for each track. Each method favors a slightly different strategy. For one hypothesis $\mathcal{H}$ we consider $S(\mathcal{T}_n)$ the highest score of track $\mathcal{T}_n$, and $L(\mathcal{T}_n)$ its length. Consider T the number of tracks in hypothesis $\mathcal{H}$. The following methods are given:

- *Simple average:* Equation 2.7 where all tracks, regardless of their length are weighted equally. This strategy works for most cases.

$$\frac{\sum_{n=1}^{T} S(\mathcal{T}_n)}{T} \tag{2.7}$$

- *Length weighted average:* Equation 2.8 where tracks are weighted by their length. It is a proper average since it is divided by the total number of observations in the hypothesis. It works best when longer tracks are preferred, but small tracks should be given time to grow. Essentially the score of the hypothesis hinges mostly on the longer tracks, leaving smaller tracks to grow before they start to seriously contribute. This method also works very well when combined with a fixed maximum track length.

$$\frac{\sum_{n=1}^{T} S(\mathcal{T}_n)L(\mathcal{T}_n)}{\sum_{n=1}^{T} L(\mathcal{T}_n)} \tag{2.8}$$

- *Sum of scores:* Equation 2.9 When simply averaging scores, hypotheses have a tendency to form very few tracks that score high. New tracks barely get a change to grow and usually only two or three objects are actually tracked accurately. To avoid this artifact it is sometimes beneficial to favor a larger number of tracks, regardless of their specific scores. By simply adding all the scores from all the tracks there is a tendency to form many useless tracks, however; therefore this

method should be accompanied by a solid track pruning strategy. For example pruning each hypothesis down to a fixed number of tracks, or removing all tracks that score below a given threshold. (More about track pruning later.)

$$\sum_{n=1}^{T} S(\mathcal{T}_n) \tag{2.9}$$

- *Weighted sum of scores:* Equation 2.10 Same as above, except favoring long tracks. Once again, it is prudent to have a solid track pruning method in place in order to use this scoring strategy. Additionally, since track lengths are considered, it is likewise important to prune observations and therefore constrain the length of the individual tracks.

$$\sum_{n=1}^{T} S(\mathcal{T}_n)L(\mathcal{T}_n) \tag{2.10}$$

- *Average weighted sum of scores:* Equation 2.11 This strategy works very well in busy and noisy environments. Most models tend to assign scores based on the last two observations in a track only, although longer tracks often indicate that a process was tracked successfully for a while. This strategy therefore offers the benefit of considering the length of a track while still averaging by the total number of tracks.

$$\frac{\sum_{n=1}^{T} S(\mathcal{T}_n)L(\mathcal{T}_n)}{T} \tag{2.11}$$

Looking at Figures 2.2 and 2.3 it becomes clear that the number of tracks and hypotheses explodes exponentially with every observation that comes in. It is necessary to do some hypothesis and track pruning to keep the size of the hypothesis-set under control. Starting at the highest level, which hypotheses should be thrown out? Recall that the score of the hypothesis reflects how well a hypothesis describes the observed world. Several options have been explored:

1. Prune all hypotheses that have a score below a set threshold. This is a straightforward, deterministic way of tossing out any hypotheses that fail to score above the threshold. Although effective, sometimes the remaining hypotheses tend to be very similar. This is not hard to imagine; if a hypothesis has several tracks that all score very high it will drive up the score of that hypothesis. Now when a new observation comes in, that hypothesis is cloned several times and all its offspring will score high as well, because of these high scoring tracks. Other hypotheses not containing this successful set of tracks will be at a disadvantage and run the risk of having all their offspring pruned when the next observation comes in. In this way a set of high scoring tracks can dominate all hypotheses. This characteristic is referred to as the *"recurring track"* problem.

2. Deterministically prune the lowest scoring hypotheses until a set number of hypotheses are left. This method first sorts the hypothesis by score and then starts removing the lowest scoring ones until a given number of hypotheses remain. This method also suffers from the recurring track problem. Depending on the threshold this method may give worse tracking performance. Especially when the maximum number of allowed hypotheses is set low, the recurring track problem can become quite severe. The benefit of this approach, however, is that the processing time required for each observation and the total memory footprint are very constant.

3. Let the hypothesis score be the *chance* that a hypothesis will *not* be pruned. This means that higher scoring hypotheses have a better chance of surviving the pruning than the lower scoring hypotheses. This method battles the recurring track problem quite well. This process can be followed until a set number of hypotheses are left, or simply a one-shot pass where there is no maximum number of hypotheses defined.

4. Track the parental history tree of each hypothesis. For any given two hypotheses investigate how long ago they had a common ancestor. The closer their relationship the smaller the chance that they will *both* make it into the next hypothesis-set. This is a generic way of battling hypothesis-sets in which most hypotheses are very similar. Intuitively it is good that many of the hypotheses are similar, because it would mean that their tracks are accurately tracking a given number of objects, however, the problem seems to be that sometimes new tracks barely get a chance to grow. Biasing hypothesis scores based on parental history forces a certain level of diversity into the hypothesis-sets which may be desirable. No extensive experimentation has yet been done with this technique, however.

Often it requires some experimentation to find the correct scoring and pruning method for hypotheses. The combination of hypothesis scoring and pruning is generally referred to as: *hypothesis control* and is of vital importance to make process detection both scalable and functional.

2.2.5 Track Pruning

As a PQS is running it will form strong tracks for objects that are correctly tracked, however, most hypotheses will also collect short, low scoring tracks. These lower scoring tracks are often a characteristic of noise, and do not indicate the tracking of an actual process. Over time many of these noisy tracks will be created, collecting all observations that could not be properly matched with a tracked process. It is therefore important that these tracks are frequently pruned away.

However, track pruning is a little bit more tricky because we have to deal with the possibility that a short, low scoring track is going to grow and become the evidence for an important process. If such a track is pruned too soon the process may go undetected. In other words, small tracks must get a chance to grow. Intuitively it would be tempting to simply not prune tracks below a set length. This will effectively exempt short (and

often low scoring) tracks from pruning, thus allowing them a chance to grow. However, this also allows the opportunity for a hypothesis to get flooded by very short (one or two observation) tracks, with a very low score. We refer to this characteristic as the "small tracks" problem. Often the small tracks problem can lead to a very poor overall hypothesis score regardless of the quality of the higher scoring tracks. There are two ways to mitigate the small tracks problem: 1. Allow only a set number of tracks in a hypothesis, and prune short tracks anyway if they score the lowest, and 2. Write better models. The second option is a usually a catch-all for most tracking problems, and that is because it is true. Often it is possible to significantly improve tracking performance by merely adjusting the way the a track is scored. (One notorious example is described in more detail below in Section 2.5). The general advice on tackling the small tracks problem is to first set a maximum to the total number of tracks allowed in a hypothesis. If that does not fix the problem, try improving the model by setting constraints on how tracks are scored. Track pruning options:

1. Prune all tracks with a score below a set likelihood. This method is straightforward and works very well in most cases. The model has to account for scoring new and small tracks that have good potential such that they will not get pruned right away. This means that new very short tracks (one or two observations) can get a score that is higher than the pruning threshold to allow them a chance to grow. Once more observations are added to that track the score can be increased or decreased, depending on how well the track represents the modeled process.

2. Prune all tracks with a score below a set likelihood but with at least a minimum length. This is the same paradigm as above except that a track must be at least of the minimum length before it may be pruned. The models can often be less complex because small tracks get an opportunity to grow before they run the risk of getting cut. Therefore the model builder does not have to account for short tracks that may in the future grow, but are currently not yet a good indicator of the modeled process. This track pruning method may suffer from the small tracks problem.

3. Sort tracks by likelihood and prune them deterministically from the hypothesis until a fixed number of tracks remain. This method can be used by itself, or combined with the two other methods above. In either case, the tracks are sorted by score and pruned starting at the track with the lowest score until a fixed number of tracks are left in the hypothesis. This method works very well in reducing the effects of the small tracks problem.

Once again the right choice depends on the specific scoring characteristics of the models used. At this point the choice of pruning techniques is done by experimentation.

2.2.6 Observation Pruning

Most domains do not require the observations to linger around in a track forever, however, only the most recent observations are important to the models. In this case it is

possible to remove the tail-end of a track and throw out the older observations that no longer have any significant impact on the score of the track. Although this method has no impact on tracking performance and accuracy, it does (significantly) improve computational performance. If older observations are pruned from prominent tracks the total memory footprint of the hypothesis-set is going to be steadier and under control. This leads to better computational performance as well, since all the work is done on a smaller memory footprint.

For example, imagine the tracking of an airplane through a large airspace. If a radar system gives 1 observation per second and the airplane is within radar range for several hours, the track associated with this airplane could easily grow to over ten thousand observations. However, any reasonable model would only use the last three or four observations to track the plane. If the tracks are kept short (say no more than 10 observations) the score of the tracks would still be the same (because it is determined based only on the last 3 or 4 observations) while the memory footprint is significantly reduced. Methods for observation pruning:

1. Prune observations by track length. This is the most straightforward method which prunes older observations until the length of the track is at a fixed maximum. This method implies that as new observations are added to a track, older observations are pruned out. The maximum track length must be at least as high as the maximum number of observations that *any* model in the tracker looks at. (Clearly it would be unwise to prune observations from a track that a process model still might need to evaluate).

2. Prune observations by age. This method works best when observations come in at a relatively steady rate, or when models use observations based on their age. For example, if the process models only use observations that are newer than 30 seconds, then it is safe to prune any observations that are older than 30 seconds or so.

When pruning observations it is important to keep the hypothesis scoring methods in mind. For example, consider using a hypothesis scoring method that multiplies the track score by the track length; then emphasis is given to longer tracks. If the tracks are kept too short by the observation pruning, then the hypothesis scoring method will not function properly. A common artifact that happens in such a case is when there is a very distinct noisy sensor in the input, that gives the same observation at a very high frequency. If observation pruning is not done by *age* then the PQS algorithm will have a tendency to start multiple tracks for these noisy observations, splitting the observations between these tracks. Needless to say, this is a method of keeping hypothesis score artificially high. (Generally this artifact can easily be removed by pruning observations by age and using a track-length-based hypothesis scoring algorithm, however, in the most severe cases a modification to the model must be considered.)

2.2.7 Track Score Decay

While experimenting with the initial implementation of a Process Query System it soon became clear that old tracks have a habit of lingering around statically. When an event would occur and a process model would detect it, it would score a track very high. After the event would be over the track would not gather any new observations and would simply hang around forever. This would prevent other tracks from growing and the hypothesis would not be representative of the observed environment. The straightforward solution to this problem was to decay the score of a track when no new observations are added to it. Since a track is evaluated by every model in the PQS tracker, a vector of scores is associated with each track. Therefore the score decay must be done on the entire score vector, however, the rate of decay will be dependent on the specific processes modeled by the models.

For example, assume once again that a Process Query System is tracking airspace. Assume also that it is configured with three process models; one for personal propellor aircraft, one for commercial passenger airliners, and one model for supersonic jetfighters. The radar system reports once every five seconds on all objects observed in the airspace. Because the radar system is relatively slow the models must make predictions of where all tracked airplanes are. These predictions will be based on the last several observations in each track, where a track represents one tracked airplane. When an airplane flies out of radar range no more observations will be received and the predictions will become less and less accurate as time passes. However, the accuracy of the predictions degrades differently for different airplanes. Because of the speed and maneuverability of the supersonic jetfighters the predictions will be poor very quickly, although the propellor driven personal aircraft are so slow that the model can keep making accurate predictions for a while to come.

The above example demonstrates that when a PQS tracker is configured with multiple models, they each may have different rates at which track scores decay. There are two options for track score decay:

1. Models take care of decay. Every time an observation comes in, all models are called for all tracks in all hypotheses. This means that even though most tracks do not get extended with a new observation, the models will still be invoked for them. The model can therefore detect that a track has not been modified for a longer time and degrade the score of the track. Models can, for instance, insert extra states or transitions to govern their own decay curve.

2. PQS tracker takes care of decay automatically. This method simply makes the models less complex. The model configures the rate of decay in the tracker, for all scores associated with this model. The tracker then automatically applies the decay to all tracks. The model now only has to assign a score when a track was actually modified (i.e. when a new observation was added). This method uses logarithmic decay only.

The function for logarithmic decay is:

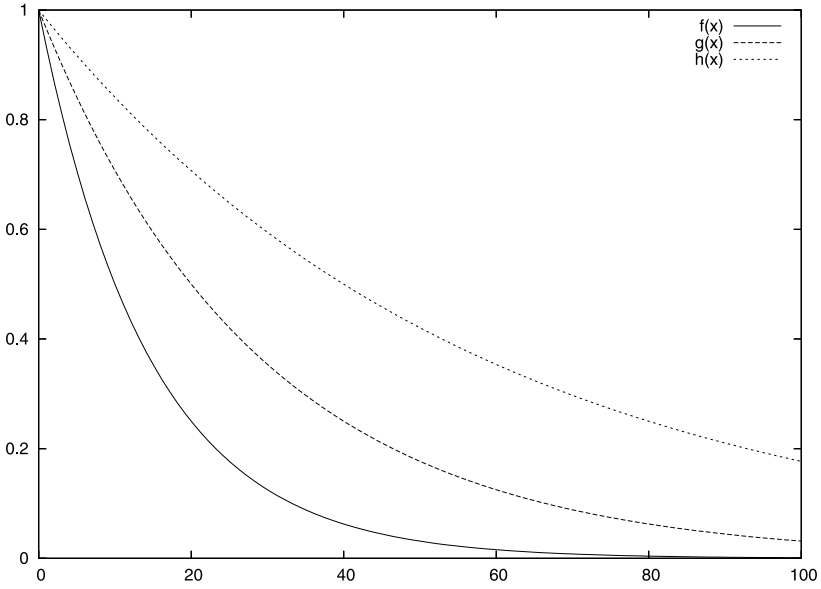


Figure 2.4: Halflife curves for $\lambda = 10, \lambda = 20$, and $\lambda = 40$.

$$s_{t_1} = s_{t_0} \times e^{-\alpha \times (t_1 - t_0)} \tag{2.12}$$

where α dictates the rate of decay. It is common to specify logarithmic decay in halflife values, where halflife value λ indicates the amount of time after which the score will be exactly half. With $(t_1 - t_0) = \lambda$, this gives:

$$e^{-\alpha \times \lambda} = \frac{1}{2}$$

$$-\alpha \times \lambda = ln(\frac{1}{2})$$

$$\alpha = \frac{ln(2)}{\lambda} \tag{2.13}$$

Halflife decay has become a fundamental part of Process Query systems. Figure 2.4 shows halflife graphs for various values of λ.

2.2.8 Model Invocation

So far the discussion focussed on the structural underlying aspects of Process Query Systems. Now we will focus on the most important part of any PQS: the actual "process

queries", or "models". Incoming observations are automatically grouped in all possible configurations of tracks in a large group of hypotheses. Models then evaluate all those tracks and determine a "score", indicating how well a track forms evidence of the queried process. The only thing not discussed so far is exactly when a model is asked to evaluate a track, and how it goes about doing that.

In principal, a model (or process query) is a simple function that takes a track as the argument and produces a score as output: $s_k = S_k^{\mathcal{M}}(T_k)$. Preferably the score is between 0 and 1, however there is no rule that explicitly requires that. The model is asked to evaluate one track at a time, meaning that the PQS tracker will invoke the model once for every track in a hypothesis for all hypotheses. The model therefore only has to focus on "how well *this* track is evidence of the process". Therefore, the model writer only needs to focus on the actual model, and does not have to be concerned with observation bookkeeping, hypothesis control, and other related matters.

The PQS system will invoke the model according to the following rules:

- For every new observation the tracker invokes all the models on all the tracks in all hypotheses, before the pruning step is done. Since a hypothesis is cloned for every track such that the observation can be tried in every possible configuration with all existing hypotheses, there will be many tracks that did not change since the last time the models were invoked. The model, however, is free to re-evaluate the track and re-adjust the score because time has passed (see the discussion on halflife times).

- Invoke the models at any other convenient time. The tracker can invoke all the models on all the tracks in all hypotheses based on reasons other than the arrival of a new observation, however, this is not a requirement. It is good practice to invoke the models at regular time-intervals to allow the models to downgrade tracks if appropriate. Since models may be computationally intensive it is recommended that the tracker watches system CPU load and try to make the periodic invocation of models coincide with moments of low load.

All models must be invoked when a new observation is added, specifically before the pruning step. Invoking at any other time is not required and is up to the PQS tracker. Additionally, it is very common for several hypotheses to contain exactly the same track. An efficient PQS implementation is therefore encouraged to invoke the models only once and copy the scores.

Although this method of computing track scores is very computationally expensive, experimentation has shown that reducing model invocation rates drastically increases the explosion of hypotheses. The reason is that pruning can only be done after model invocation. Additionally, tracking performance suffers in that case. As mentioned before, PQS tracking is a heuristic process; missed observations, sensor error margins, and exponential hypothesis explosion control (pruning) all contribute to make the "perfect" assignment of observations to processes very difficult. Technically, if all observations were received correctly, without error, and in chronological order, it is still not guaran-

teed that a "perfect" answer is computable. More about computability and performance in section 2.6.

2.2.9 The output of a PQS

The input of a PQS are observations (in arbitrary form) that come from one or more sensors. The format of these observations is irrelevant and can differ from sensor to sensor, and may even be different for observations from the same sensor. Next, it is up to the models to handle the various fields in the different observations and assign scores to the tracks that the PQS builds from the observations. Although the incoming observations and the process models are all that are needed to successfully do process detection and tracking, the system will need to publish some output as well. Ideally the output of a PQS is formatted such that it can be used as input observations for a second-level PQS. This essentially means that the conclusions published by a PQS can be used as observations for another PQS, conceptually allowing us to chain multiple Process Query Systems. The benefits of chaining multiple PQS engines become obvious when we consider splitting a complex tracking problem into multiple tracking tasks. For instance, consider tracking a flock of birds. The first-tier tracker would process a radar or video image, trying to identify the paths of individual birds. This is a very difficult problem since the observations will be too sparse to make a clear distinction between birds. However, a second-tier tracker could take output of the first-tier tracker as input and come to more general conclusions, such as the direction of the flock, its speed, and the approximated number of birds.

The two questions that remain are: (1) What does such an "output conclusion" look like, and (2) When are they published? Intuitively it seems prudent to publish only one hypothesis, since multiple hypotheses do not necessarily have meaning as output (let alone clear up any confusion as to which observations are evidence of what process). Additionally, the highest scoring hypothesis should be published, because, at the current time, it is the best description of the observed environment given the process models. It must be said, however, that in some application fields it may be meaningful to publish more than just the highest scoring hypothesis. This decision is up to the user and should be a PQS configurable option.

So what exactly should be published from the highest scoring hypothesis? The answer lies in the definition of a hypothesis. Since a hypothesis represents a complete view of all the evidence of the individual processes that are occurring in the observed environment, evidence of each occurring process should be published. So, if a PQS is tracking the movement of three balls bouncing around in a box, the hypothesis will have three tracks, each representing a collection of observations associated with one ball. The tracks, therefore, tell us which observations belong to which ball, and thus describe the paths that each of the three balls followed. So in this case the PQS tracker would publish three conclusions, one for each track (ball).

The contents of the conclusions should be up to the model. Depending on what the model was designed to do, the contents of a conclusion may differ. For the example above, we can envision a model that was made to predict the position and future path of

a ball. Such a model would publish a momentum and maybe one or two expected future locations. However, if the model was designed to report the path of the ball for the last 10 seconds, then the published conclusion should contain a list of past locations for the ball. What this means is that the model should determine the contents of the conclusions. Notice, however, that both examples refer to relatively static data associated with the track. We could, for example, store a data section $\mathcal{T}_d^{\mathcal{M}}$ with each track $\mathcal{T}$, for every model $\mathcal{M}$ in the system. A model would then be free to use this data section as a canvas to store data with the track, eliminating the need for the model to recompute all these factors every time the model is invoked for a track. This datasection, then, holds the state of the process modeled by $\mathcal{M}$. The predictive model, for example, would store the momentum with the track and would then only have to update the momentum every time the model is invoked. The model would also update the predicted future locations of the ball in this track-specific data section. Now, when the tracker decides it is time to publish this track, it only needs to publish the track-specific data section $\mathcal{T}_d^{\mathcal{M}}$ without having to invoke the model again. This effectively separates the model evaluations and the publishing tasks. Therefore the model only has to update its track-specific data section for each track that it evaluates, and the PQS tracker will simply publish this section as a conclusion when necessary.

Now let's assume that there are two models, $\mathcal{M}_1$ and $\mathcal{M}_2$ in the tracker. This means that some tracks will score high for the first model $\mathcal{M}_1$, and others will score high for the second model $\mathcal{M}_2$. This also means that each track will have *two* track-specific data sections; $\mathcal{T}_d^{\mathcal{M}_1}$ and $\mathcal{T}_d^{\mathcal{M}_2}$, one for each model. It would be proper to only publish the data section for the model that scored highest for the given track. It should be clear that publishing anything else would be against the principle of process detection. A sequence of observations is usually only indicative of one occurring process, therefore, the model that matches this process best will ultimately dictate the score for that track, and with that, the conclusions.

Now that conclusion creation and conclusion publication are functionally separated, the publishing thread of a PQS tracker can autonomously make decisions on when to publish the conclusions. The publication frequency will depend on the specific requirements that the PQS is deployed for. If the observation frequency is very high, the publication frequency might be best set lower, however, if the observation frequency is very low, models can make predictions which can be published at a higher frequency. Examples of these conclusions will follow in the Process Query Modeling Language sections 2.2.10, and 2.2.11.

2.2.10 The Process Query Modeling Language

From the specification document: (Attached in Appendix A).

> The Process Query Modeling Language (PQML) is an assembly-inspired, low level language specification for defining observations and models for Process Query Systems. Any implementation of a Process Query System

must be able to parse PQML (pronounce Puh-Que-Mol) files and obtain observation layouts and model definitions from such a file.

There is no explicit information given on the binary representation of a PQML definition, it is up to the implementer of the specific PQS to determine what the best internal representation for its PQS is.

It is expected that all higher level models can be compiled into a PQML program and so be used by a PQS. Hidden Markov Models, or state machines can have their transition predicates be PQML programs, thus making the translation to PQML more convenient.

PQML is meant to be the lowest level specification for PQS models. However, it is still fully platform independent since it does not specify a binary representation (although there are minimum standards set for all constructs). Essentially PQML defines a set of instructions and structures that represent all fundamental (atomic) operations that a model can perform in a PQS. The goal in designing PQML was therefore to represent all these fundamental operations in as simple a language as possible. This means that all other models can compile to PQML and still be fully functional. For efficient model building it is recommended that the model builder use a higher level model specification method. Sensible modeling methods include Hidden Markov Models, Petri Nets, rule sets, and iterative programs.

To fully appreciate PQML it is important that the exact function of models in a PQS is clearly understood. Recall that the PQS tracking core takes care of all the hypothesis and track creation, and pruning. Models are invoked after new observations arrive, and in between observation arrivals. Models are therefore often invoked on tracks that have not changed, but only aged instead. *The task of a PQS model is to assign a score to the given track, based on the observations that the track consists of, representing how well the track is evidence of the modeled process.* Thus, a model $\mathcal{M}$ is invoked with a track $\mathcal{T}_k$ at time k as input, the model then returns a score $S_k^{\mathcal{M}}(\mathcal{T}_k)$ representing how well the observation sequence $O_{1\ldots n}$ of $\mathcal{T}_k$ fits the definition of the process that model $\mathcal{M}$ is modeling.

The models are executed on a virtual machine which is reset every time a model is invoked. Special instructions are available to retrieve data from observations in a track, and to set the score. The virtual machine is register-register driven with separate data and instruction memory (however, there are two separate types of data memory; 1. persistent with each track, i.e. process state, 2. globally persistent.) There are three basic datatypes in PQML; integers, floats, and strings. Integers are at least 32-bits signed, floats are IEEE Standard 754 compatible, and strings must be able to contain at least 32 characters (including any terminating NULL "\0" characters). There are at least 32 registers of each type, named $i0\ldots i31$, $f0\ldots f31$, and $s0\ldots s31$. For the data memory (both track specific, as well as global), and for the observations it is possible to define arrays for both integers as well as floating point data. Arrays are indexed with special array instructions. Finally, there is a stack to and from which all three datatypes can be pushed and popped. This stack is also used to store return addresses for function calls.

The seemingly limited number of basic datatypes in PQML was a deliberate decision to keep the language as simple as possible. The goal was to evaluate the language as is, and grow its capabilities as needed. Specifically, any changes in data type directly affect the way in which the PQS system communicates observations and conclusions over the network. So far no models have had a need different from what is outlined in PQML. Additionally, PQML does not feature a pointer datatype. Although integers can be used to index arrays, it is not possible to store, for example, a pointer to an observation. The reason for this choice is not only the added complexity that this would bring, but also the fact that observations are not unique structures, and are internally shared amongst tracks. One future extension that is under consideration is to mitigate the current inability of the models to access the observation "fidelity". Each observation is given a fidelity number $f = 1 - e$, where e is the error margin in the observation as reported by the sensor. If this feature is added, a model will be able to assess the confidence that a track is correct, given the error in its observations. Another possible future addition is a *mayMatch* function that returns a yes-or-no answer indicating if an existing track and a new observation could fit together, effectively avoiding creating useless tracks and hypotheses.

A PQML program is split up in "program sections". These program sections must be one of six types, however may re-occur as often as needed:

1. *data* Defines the globally persistent data. Arrays may be used in this section. This is also the section to define constants because PQML does not support immediates in its instructions. Variables in a *.data* section can be used for dynamic parts of a model, meaning that these variables are global. Every subsequent invocation of the model can read and write to these variables. For example, it could be used to count how many times a model was invoked. More practical uses include keeping track of the layout of a network; a special observation updates the model's internal network graph, which is kept statefully in the *.data* section. Also, this section is very useful for models that can be trained.

2. *observation* This section defines the layout of one incoming observation type. It also requires an identifier name. There can be many different observation types that a model can handle, so there are likely to be several *.observation* sections in each model. Array definitions are permitted, keeping in mind that the array size must correspond to the producing sensor and this model definition. Additionally, each observation carries with it a timestamp which can be retrieved with the *lobsts* (load observation timestamp) instruction. This timestamp is assigned when the observation was received in the PQS tracker.

3. *conclusion* Defines the output of the model. There may be only one *.conclusion* section in every model. It is also the track stateful data section for this particular model $\mathcal{T}_d$. What this means is that the variables in this section stay with a track and are specific for that track. This section is published by the PQS tracker when the track is selected for publishing. This section is very useful for storing the state in a stateful model. For example, in kinematic tracking it can be used to store the momentum of the tracked object, and also one or more predicted next positions.

4. *text* The actual model. Text is the code that gets executed when the model is invoked. Unlike labels in the various data sections, labels in the *.text* sections are followed directly by a colon ":". Execution of the model starts at the *start:* label and ends when the *exit* instruction is encountered.

5. *halflife* This directive sets the speed with which track scores will decay over time. After the specified time (in seconds, floating point) the score of the track associated with this PQML model will have decayed to half its original value, unless it was overwritten by a new *setl* instruction. Disable by setting a negative value.

6. *include* Includes another PQML file. The specified file may contain all other sections, including *.include* sections. It is the task of the programmer to ensure that includes do not redefine labels or contain cyclic inclusions. This feature is extremely useful for providing libraries of commonly used functions.

Notice how conclusions and observations are defined very similarly. The intention is to allow the output conclusions of one PQS tracker to be input observations of another. In this way trackers can be chained. For example, in military command and control it is often necessary to track movement of entire groups of units. A PQS tracker can then be configured with several PQML model for identifying troops, vehicles, tanks, and jetfighters. The conclusions from all those models would then include: type of unit, current position, direction, and speed. A second level PQS tracker would take these conclusions and could be configured with a PQML model to associate groups of units. This second level PQS tracker would then be tracking group movements.

The inputs of a model are the observations, the outputs are the track-specific conclusions. This defines a many-to-one relationship. A model can take input from various sensors (or other trackers) and have a range of different input observation types, however, it may only publish one particular conclusion observation. Theoretically there is nothing that forbids looping: the output conclusion is used as an input observation for the same model. No experiments have yet been performed with such a configuration.

A second feature to notice is the difference between globally accessible variables (in the *.data* sections) and the track specific stateful data (in the *.conclusion* section). Note that the track specific stateful data is also the conclusion that is published by the PQS tracker when *this* PQML model assigned the highest score to a track. This section is also very useful to store intermediate data that would otherwise require the model to completely re-evaluate the track. The global *.data* sections have proven to be very useful in self-calibrating models.
(For more specifics on PQML please refer to the PQML specification document.)

2.2.11 A PQML example

This section introduces a PQML model that takes observations containing only an integer number. The model sorts these observations into tracks, each track taking observations for one integer number. Effectively the observations are thus sorted by content. The published conclusion gives the numbers for which tracks were made, and how many

observations there are in each track. Note that this is merely an example PQML model, although it shows most of the concepts discussed so far.

The first section of this program defines the halflife time (10 seconds), the format and identifier name of the incoming observation and the format and name of the outgoing conclusion, which is also the track-specific stateful data section. The scores associated with this model for every track will decay to 50% of their given values in 10.0 seconds, unless this model overwrites them. Only one type of incoming observation is defined, named *input* which only contains an integer field, called *number*. The default value for this field is 0, however, the actual value is not determined by the PQS system, but by the sensor or the system that produces these observations. The conclusion named *output* is the stateful data that is stored with each track. Also, when conclusions are published, this is the section that gets sent out, assuming this was the highest scoring model for the track.

```
;
;  PQML example
;  Vincent Berk
;

    ;  Decay tracks to 50% over 10 seconds

.halflife 10.0


    ;  The incoming observation contains a number
    ;   TYPE    LABEL     DEFAULT_VALUE

.observation input
    int     number          0


    ;  The conclusion just keeps track of what the
    ;  number is of the track, and how many observations
    ;  there are in this track.

.conclusion output
    int     type_number     0
    int     count           0
```

The second code block defines the global data section. There may be multiple *.data* sections defined in a PQML program, as long as there is no overlap in label names. All variables in this section are global and may be modified. These modifications are permanent and will persist between model invocations. The variables that have the prefix: *const_* are considered constants for the programmer, however, this is not a language feature. Since PQML does not feature immediate values in the instructions it is necessary to store all known values as global variables. It is up to the programmer to ensure that these variables do not get modified.

```
    ;  Globally stateful data.
    ;  Defines some constants and keeps track of how
    ;  many times this model has been called.

.data
    int     model_calls     0
    int     const_0         0
    int     const_1         1
    float   const_zero      0.0
    float   const_half      0.5
    float   const_one       1.0
```

The third block is the beginning of the actual model code. Processing begins at the special *start:* label. The first instruction, *tmod* lets the model know if the track was actually extended with a new observation since the last time this model was invoked. Using this instruction a distinction can be made between the arrival of a new observation in this track, or merely the passage of time. In this particular case, if no new observations were added, the model simply jumps to the end. If a new observation was added, the model checks the size of the track to determine if this is a totally new track (size=1) or if there were other observations in this track before the new one got added (size>1). The *ALL* directive specifies that the model wants to retrieve the length of the entire track, instead of only the number of one type of observation in the track. For example, it is possible to simply query the number of observations of type *input* by calling *ldsize i0, input*. However, since this model only specifies one type of incoming observation it is safe to use *ALL* here.

```
;
;  The actual model code:
;

.text
start:
            ;  If no new observations were added since
            ;  last time this model was called, then
            ;  just go to the end

        tmod
        be          end

            ;  Get the track length.  If it is 1, then
            ;  the track is totally new.  If longer
            ;  than 1, it must be checked for a match.

        ldsize      i0, ALL          ; get the size
        setci       i0, count        ; set conclusion

        geti        i1, const_1      ; get 1
```

```
        cmpi        i0, i1            ; is the track size 1?
        bne         extend_track      ; jump if not
```

The fourth block handles tracks that are totally new and never have been evaluated before. The previous code block ascertained that the track has not been evaluated since the new observation was added. It also assures that the track only has one observation; the new one. Therefore this block of code goes ahead and gets the *number* of the newest observation in the track (indexed by a zero in register *i0*). It then stores this number in the track stateful data section, specifically in the variable *type_number*. This field will be published as a conclusion if the PQS tracker decides to publish this track. From now on only observations with the same number in the *number* field will be allowed to be appended in this track, as we shall see in the fifth code block further down. The second thing that is done in this block is setting the score (sometimes referred to as the "likelihood") to 0.5. This is a new track and the likelihood is deliberately kept lower than the likelihood for older and longer tracks. The reason is to discourage new tracks to form if there already is an existing track with the same *type_number* in the hypothesis. A descriptive example will follow further down below.

```
            ;  else fall through:

singleton_track:

            ;  There is only one observation:

        geti        i0, const_0
        lobsi       i2, input[i0].number   ; get from obs
        setci       i2, type_number

            ;  Set the track score to 0.5

        getf        f0, const_half
        setl        f0                ; set the score

            ;  done

        jmp         end
```

The following code block (fifth) deals with tracks that have more than one observation, and just had a new observation added. These tracks already have a *type_number*. It is therefore necessary to retrieve the type number from the track-specific stateful data section (using the *lobsi* – load observation integer instruction) and compare it to the *number* in the newly added observation. If it is a match then the track gets a very high score (1.0), if it is a mismatch then we must discourage this track and hypothesis as much as possible (score will be 0.0).

```
extend_track:
```

```
            ;  There are multiple observations:
            ;  Get the number of the last observation,
            ;  then get it from the track, and compare.

        geti        i0, const_0
        lobsi       i2, input[i0].number  ; get from obs
        getci       i3, type_number       ; get from track
        cmpi        i2, i3              ; compare
        be          set_one             ; if equal set score to 1

            ;  Set the track score to 0.0

    set_zero:

        getf        f0, const_zero
        setl        f0                  ; set the score
        jmp         end

            ;  Set the track score to 1.0

    set_one:

        getf        f0, const_one
        setl        f0                  ; set the score
```

The last code block is the end of the model. To update the statistics, it retrieves the global variable *model_calls* and increments it before writing it back. This variable will count the number of times the model is invoked by the PQS tracker. In order to actually publish this number it would have to be written to a variable in the track-specific data section, because those are the only variables that ever get published. Finally, the program finishes by calling the *exit* instruction.

```
end:
            ;  Increment the number of times this model was
            ;  called before exiting.

        geti        i0, model_calls     ; get global variable
        geti        i1, const_1         ; get a constant
        addi        i0, i1              ; add them up
        seti        i0, model_calls     ; write global var back

        exit                            ; exit the model
```

Now lets consider an example situation to show how the model performs in a PQS tracker. Given in Figure 2.5 is a hypothesis with two different tracks, one with observations with *type_number* 5 (blue), and another one with *type_number* 3 (red). At time $T = 1$ there are three observations in the blue track (*type_number*=5), and four in the red track (*type_number*=3). Due to the halflife decay, blue has a score of 0.8, and red has a

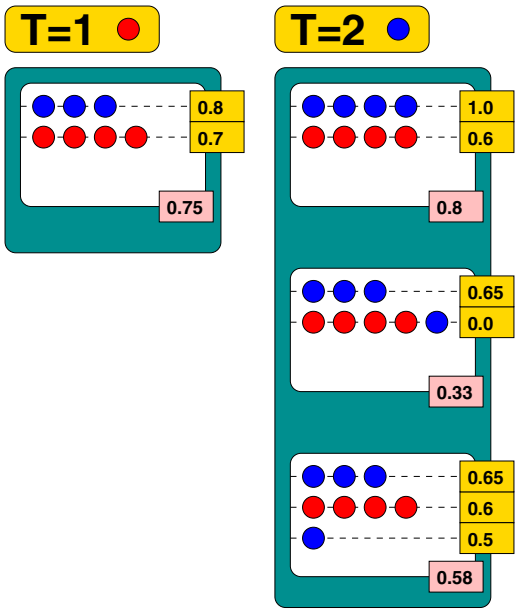


Figure 2.5: Deterministic sorting with a PQS.

score of 0.7. At time $T = 2$ a new observation arrives with *number*=5 (blue). Given the single hypothesis at time $T = 1$, three new hypotheses are generated; one where the new observation is added to the existing blue track, one where the new observation is added to the existing red track, and one where the new observation is in a track all by itself.

So, according to Figure 2.5 that generates three hypotheses with a total number of seven tracks. For simplicity we will assume that the PQS tracker is configured with only the PQML model described above. The model will therefore be invoked seven times because of the arrival of a new observation, at time $T = 2$. The first hypothesis at $T = 2$ has the new observation added to the blue track. This is the ideal situation, and is what we were looking for. The model assigns a score of 1.0 to the blue track. The red track has not changed and the model will therefore not assign a new score to it. Notice, however, that the score on the red track has decayed due to the halflife time set by the model. The averaged score of this hypothesis is 0.8.

The second hypothesis adds the new observation to the red track. This is clearly a mismatch between the track *type_number* (3) and the *number* of the new observation (5). The model will therefore assign a score of 0.0 to this track. The blue track is not modified and the model will therefore not assign a new score. Halflife decay brings it down to 0.65. The average score of this hypothesis is 0.33; very low, because it contains a track which is undesirable.

Finally, the third hypothesis leaves the two existing tracks untouched (notice the halflife decay on their scores). A new track is created containing just the newly arrived

observation. Although this does not necessarily violate the rules of the model, it still is an undesirable situation. If the new track would get too high a score there is a risk of it overtaking a more desirable hypothesis (the first one). So, for this reason new tracks must get a lower score (0.5 in this case) which leads to a 0.58 score for the hypothesis, placing it well below the first hypothesis in this hypothesis-set. To ensure that this model will perform deterministic sorting we must prune all hypotheses but one. In a less deterministic situation it would be better to keep multiple hypotheses around.

2.3 Model Building Hints

Building PQS models is often straightforward and easy, however, fine-tuning models can sometimes be difficult. This section introduces some tips that may help improve model performance. One of the most common artifacts of poorly balanced models is the formation of many short tracks versus one long one. Although the model builder knows that there is only one process generating the observed events, the process model groups the observations into multiple tracks anyway. This suggests that there are multiple instances of a process occurring in the observed environment, which is incorrect. Other causes of this artifact include very noisy observations, or an overload of observations.

The trick to minimizing this problem is by favoring long tracks over short ones. After computing the track score, the model must multiply the score by the following formula, where x is the length of the track:

$$f(x) = 1 - \frac{\alpha}{(x+\alpha)} \tag{2.14}$$

where α configures how quickly $f(x)$ rises to 1. Figure 2.6 shows this curve for three different values of α. Picking the proper α value for a model can increase tracking performance significantly. Based on experience, as a rule of thumb it is recommended to choose α to be about half to one third of the average expected track length:

$$\frac{1}{3} \times Avg(\mathcal{L}_T) \leq \alpha \leq \frac{1}{2} \times Avg(\mathcal{L}_T) \tag{2.15}$$

When, for efficiency reasons, the PQS tracker is configured to prune old, no longer relevant observations from tracks, it is recommended that a "virtual length" variable is kept in the track-specific stateful data section. This variable can then record how long the track would have been if observations had not been pruned. It is used for x, instead of the actual track length.

The PQS tracker is at liberty to invoke the models at any time in addition to invoking on new observation arrival. This allows models to account for aging of a track, and it allows for models to update state and/or position predictions. Other models, however, do not require this and therefore do not benefit from a high rate of invocation between observation arrivals. Such models may wish to use the *tmod* instruction to determine if a new observation has arrived since the last invocation of the model. If this is not the case, the model can simply exit without any processing done. Keep in mind that setting

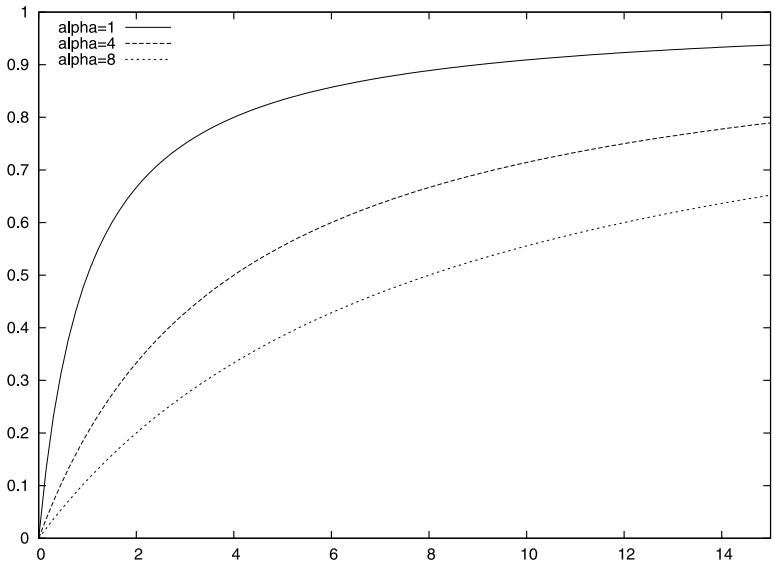


Figure 2.6: The $f(x) = 1 - \frac{\alpha}{(x+\alpha)}$ function for $\alpha = 1$, $\alpha = 4$, and $\alpha = 8$.

a halflife value will ensure that track aging automatically takes place, no new likelihood has to be set by the model.

Finally, sometimes it is useful to force a PQS tracker to act deterministically. It is, for example, very easy to make a PQS sort incoming observations into tracks, given a selective feature. Imagine tracking stateful TCP connections with a PQS to determine if any communication contains a bad signature. To match a signature it is important that all packets are re-assembled, such that the whole communication can be evaluated. (If we did not do this, a possible attacker could slip the attack past our sensor by simply slicing it up into many small packets.) So all packets from a given stateful connection must end up in the same track. This problem is very similar to the example PQML program discussed above; instead of comparing *type_numbers* we now must compare source and destination IP addresses, source and destination ports, and the protocol used. To ensure that all packets from one session end up in the same track, without two or more tracks forming, it is important to prune away any hypotheses that do not conform. This situation is identical to the third hypothesis from Figure 2.5 and can be dealt with by simply only allowing *one* hypothesis to remain after pruning. Therefore, *to make a PQS tracker deterministic, you must enforce one, and only one hypothesis to survive pruning*. This is a configurable option in a PQS tracker. (Note that this effectively configures a PQS as a

"plain old signature intrusion detection system", by using it as a stream processor.)

The opposite, increasing the maximum number of hypotheses, has a similar effect as increasing the population size in evolutionary algorithms; small low-scoring tracks have more time and opportunity to grow because their hypotheses are not pruned as aggressively. This may increase tracking accuracy environments where processes are easily confused. Therefore, when tracking accuracy is poor in complex environments, it may be worth raising the maximum number of concurrent hypotheses.

2.4 The DBMS, PQS analogy

A common question about PQS is: "Can it do ... better than ...?" Where the dots may stand for practically anything. The answer ranges from "sometimes" to "definitely", and occasionally: "no". In this section we will explain why this is the wrong question to ask. A PQS allows for the fast and easy creation of powerful process trackers. This does not necessarily mean that a PQS airplane tracker will perform better than a custom written one; however, it does mean that the time it will take to create a PQS-based tracker will be significantly shorter than the time it takes to create a custom system. This allows the programmer to spend more time on fine-tuning the models, and therefore the PQS tracker may perform better.

The analogy with DataBase Management Systems (DBMS) is therefore strong. In the pre-SQL era of databases, a programmer would spend a lot of time taking care of file handling, disk accesses, sorting, query handling, searching, etc. A whole new database system had to be built for every application. When SQL was first introduced by IBM in 1979 as the query language for the System/R database project, it allowed programmers to abstract entirely from all file management, searching, etc., and instead focus on formatting records and building SQL queries. The SQL databases nowadays are not necessarily better than custom-built databases, however, they are created in far less time. The saved time can be used to improve standard queries, the user interface, and the general usefulness of the system. Using a commercial, off the shelf DBMS to implement a customer database, library catalog, or a financial accounting system simply means creating the records and building the queries. The DBMS remains unchanged at the core, taking care of all file management, searching and retrieving and all other common database tasks. It makes building databases easier, thus allowing more powerful databases to be built in less time.

To apply a Process Query System to a particular field all that the programmer needs to do is write the proper PQML queries and run the PQS. A PQML model defines the input observation formats and output conclusion format (compare: the record format for a DBMS), as well as the process model (compare: the SQL queries for a DBMS). PQS/PQML makes building trackers easier, thus allowing more powerful trackers to be built in less time. Because a PQS is a heuristic method working in a possibly noisy and lossy environment, the analogy breaks down on deterministic base performance. Looking at an SQL query it is obvious what its result will be, and if it will work or not. The same cannot always be said about PQML queries; the performance of a PQS is determined in

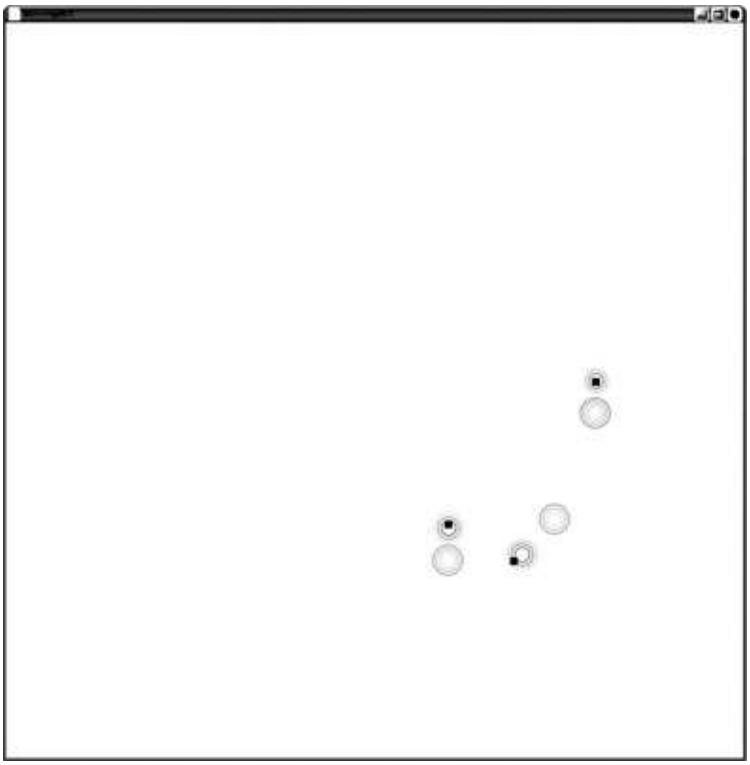


Figure 2.7: Three dots moving in circular orbits using a high publication rate.

a large part by uncontrollable factors in the observed environment.

2.5 Illustrative Example: Simple Kinematic Tracking

This section gives an intuitive example where a PQS is applied to a specific problem. The problem domain contains several dots moving around in circular orbits, at varying speeds. The positions of the dots is published occasionally, at set intervals (very much like a sweeping radar). Each published location is an observation, containing only an X and Y coordinate, and no other information. About 10% of the observations are lost and are never published. The observations are the only source of information about the observed environment. (The frequency of observation publication, the percentage of lost observations, the speed and paths of the dots are all variable.) Figure 2.7 shows the dots moving. The concentric circles indicate when and where the location of the dots was published.

The goal of the tracker is to associate observations coming from the same dot into one track and then to predict the location of dots between observations. In this way a

monitoring system can plot a smooth path for all the tracked objects, while the observation frequency is low, and/or the lost observation rate is high. The model therefore needs to correctly associate observations to be able to calculate the momentum of each of the tracked objects. This momentum, combined with the last known (observed) position can be used to calculate the predicted current position at any time.

The format of the input observations coming from the environment:

```
.observation position
    int       pos_x
    int       pos_y
```

The format of the output conclusions:

```
.conclusion prediction
    int         cur_pos_x
    int         cur_pos_y
    float       momentum_X
    float       momentum_Y
    int         object_number
```

The variable *object_number* is intended to identify the track that produced the conclusion observation. In this way, subsequent conclusions giving follow-up predictions can be associated with the same object. In the model this number can be set to an arbitrary value for every new track that gets started (*tmod* combined with a tracksize of 1). This number must be considered as the "unique identifier" for the tracked object. It is the only way outside of the PQS to establish which track was published. It is also important to note that the less rugged models will sometimes get confused when dots cross paths, and therefore start new tracks once the objects moved far enough apart again to no longer be confused. In such models the *object_number* will change after "close encounters".

In order to correctly calculate the momentum the model must assure that observations are correctly grouped into tracks. Once this is done, the momentum is simply calculated by dividing the traveled distance in the X and Y direction by the difference in time Δt. Grouping observations correctly into tracks is the actual tracking problem and lies at the core of the model. In a PQML-based PQS tracker this is done by assigning a score to tracks evaluating how well they fit a modeled process. In this domain the modeled processes are the moving dots, therefore the model score describes how likely it is that a new observation is associated with a track. The tracks are groups of observations associated with the motion of one dot, so each track follows a dot, or each track collects all the evidence for one dot. Therefore, if a track is evaluated in which the new observation is not likely to have been part of the path of the dot that this track follows, then the score of this track should be set low. However, if the new observation lies exactly in the path of the dot that this track is tracking, then the score should be high. The pseudo-code for the model:

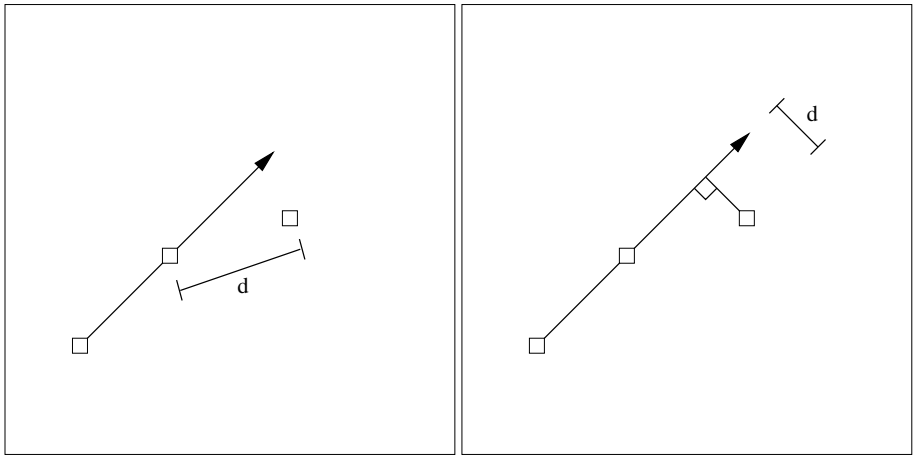


Figure 2.8: **a** (left): travelled distance d is used to determine the score; the closer together, the higher the score. **b** (right): distance d from the predicted path is used to determine the score; the smaller d the higher the score.

```
if (new observation)
    compare to assign score
    update momentum
else
    update predicted position
```

This can be done, for example, by checking the absolute distance between the last known location of the tracked object and the new observation. If this distance is too large then the score should be low, because it is unlikely that this observation was generated by the same process. This method is actually quite effective when the sampling frequency is high, although it is not very resilient to missed observations. The method breaks down quickly when two or more dots cross paths, because in that case many combinations seem valid. For an improved model we must also take the direction and speed of the object into consideration. The primary benefit of this model is its simplicity. Figure 2.8 **a** shows the way this absolute distance d is calculated.

A far better model actually takes into account the momentum and the predicted path of tracked objects. By comparing a new observation to the predicted path of the track a far more accurate score can be assigned. So when a new observation is very close to the predicted position of the tracked object the score is set very high. If the new observation deviates significantly from the predicted location, the score is set low. This method relies on the presumption that the tracked objects do not suddenly change momentum; their paths and speeds must be relatively stable. Figure 2.8 **b** shows the way this absolute distance d is calculated. In both cases the score is calculated as $1-(d\times\xi)$, where ξ represents a scaling factor. This type of model is a simplified case of the Kalman

Filter [58] (no control information is included, and the estimated error in the prediction is omitted, instead only the actual error between prediction and observation is calculated.)

Both models described above can be improved by setting limits on momentum. When the calculated momentum is higher than a predetermined threshold the score of the track is set low (near zero). This is especially effective when tracks start to form initially. When the hypothesis is mostly empty and does not contain any high scoring tracks yet, some very unlikely combinations may form; for example tracks that tie together observations that are on the opposite sides of the observed environment. The perceived momentum is then very high. These tracks can be discouraged from initially forming by limiting the momentum.

Another improvement to the last model would include the calculation of the *change of direction and speed*. If the speed is steadily increasing, or the direction is gradually changing the model can measure this and make better predictions. For the second model these predictions are important because they determine how well new observations fit the track. Better predictions therefore mean better tracking performance. Since the dots follow a circular path their direction is always changing by the same amount. If the model can detect this then the predictions will be far more accurate, therefore making the model stronger and better able to cope with missed observations.

Finally, predictions are made by the models by simply taking the momentum of the track, multiplying it by the time since the last observation, and then adding that distance to the last observed position. This is done every time the model is invoked, even when no new observations were added to the track. Figure 2.9 shows three dots moving in circular orbits. The PQS is configured with one model that calculates the momentum of the tracked objects. The models are called many times per second and may therefore publish conclusions (including the predictions) between actual observations. The dots are white boxes, and the concentric circles indicate when and where locations were sent into the PQS. The crossed markings indicate the predictions of the model for the tracked objects. Note that all markings fade out with time, thus the most recent predictions/publications are brightest. The full model code can be found in Appendix B.2.

2.6 Performance

Before applying the power of a Process Query System to any particular application domain it is useful to be able to estimate the performance of the PQS based application. This will then provide helpful guidelines on parameters and specifics of the models to be used. Although the actual performance will be very specific to the particular domain, this section aims to give some general rules of thumb to focus attention on specific considerations when using a PQS.

For a PQS the term "performance" can be taken to mean two different metrics:

- *Accuracy.* A measure of the accuracy of the tracking outcome. Given a set of parameters and a model for the PQS, how accurate is the tracking and disambiguation of the system as a whole (including the sensor network, PQS, and the model)? This

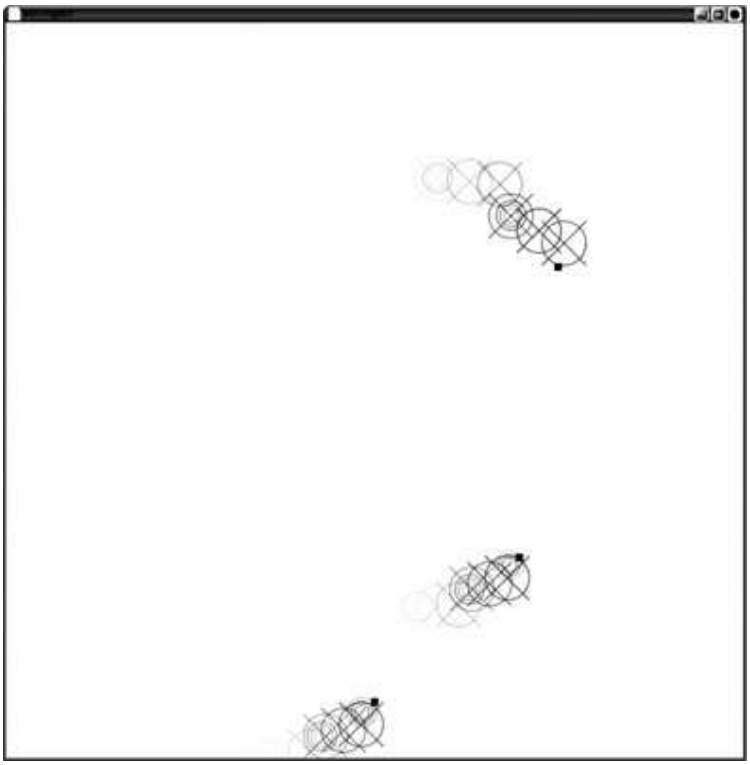


Figure 2.9: Three dots at lowered publication rate are tracked by a predicting model. The crossed marks are the model predictions between actual observations. Note that this model is unaware of the circular direction change.

can be measured by determining how many observations were correctly correlated together and how high the confidence is in the resulting hypotheses.

- *Efficiency.* A measure of the amount of work that needs to be done to reach a certain hypothesis. This is usually best expressed as the total number of observations that were handled per second, or the load on the system running the PQS core. The efficiency is for a large part determined by the configuration of the tracker and the complexity of the model.

Both analytics as well as experiments are used to learn the complexities and performance potential of a PQS. This section will point out the bottlenecks and the relative importance of several PQS parameters by showing the effect of changing parameters such as the complexity of the environment, and the number of concurrent hypotheses in the system. Finally, the section is concluded with some ideas that can drastically increase accuracy and efficiency for general cases.

2.6.1 Experimental Setup

In addition to analyzing the complexity of the PQS algorithm, a lot can be learned by experimentally changing parameters in a controlled tracking environment and measuring how the change effected the accuracy or efficiency. The specific experiment that was used involves N balls bouncing around in a two dimensional unit box ($x \in [0.0, 1.0]$ and $y \in [0.0, 1.0]$). When a ball hits a wall it turns around deterministically by inverting the x or y component of the momentum, depending on which wall was hit (see Figure 2.10). Balls do not rebound off each other.

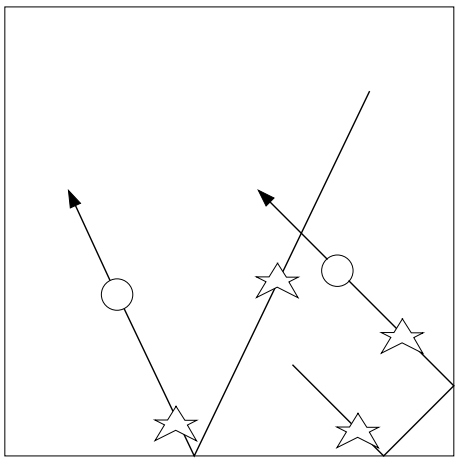


Figure 2.10: Two balls bouncing around the unit box. Observations were made at the points indicated by the stars. The ball on the left is travelling faster than the ball on the right.

All balls are assigned a random speed with both the x and the y component of the momentum being a random number between 0.0 and 0.1, thus giving any ball a possible topspeed of $\sqrt{2 \times 0.1^2} = 0.1414$. Sampling was done based on a delay Δt between measurements, meaning that an observation containing the (x, y) position of each ball was generated every Δt seconds. However, due to the inherent limitations of the computing hardware, the timing of these observations is by definition noisy since the motion of the balls, the measurements, and the creation of the observation are all asynchronous. The noise in the timing of observations is therefore an artifact, however, it only had a significant effect on the measurements at the highest of sampling frequencies with the greatest number of balls. Although this artifact could have been removed, it is actually quite representative of real world sensor environments.

The observation stream thus consists of (x, y) coordinate pairs with a hidden third component indicating the ball ID, which may only be used for accuracy evaluation after tracking. The tracker therefore only sees a stream of unlabelled coordinate pairs.

Noise and Lost observations

Besides the noise in the timing, a perturbation in the actual (x,y) coordinates can be introduced. This noise is modeled using a Gaussian distribution on distance and a flat, uniform probability distribution on angle. The stochastic variable G is Gaussian normal distribution with mean $\mu = 0$ and variance $\sigma^2 = 1$. The uniform distribution has stochastic variable U where $U \in [0.0, 2\pi\rangle$. The severity of the effect of the noise can be controlled with the variable η, taken in the range $\eta \in [0.0, 0.01]$. The deviations caused by noise are then computed as:

$$\Delta x = \eta \times G \times \sin(U) \tag{2.16}$$

$$\Delta y = \eta \times G \times \cos(U) \tag{2.17}$$

The effect of the above noise is that the length of the average perturbation is noise control variable η. For the majority of the experiments the noise control variable is set to 0, allowing only for natural perturbations in the observation timing.

In addition to the two types of noise described above, the simulator also has the option of dropping observations as if they were lost or never registered. The lost observation control variabable γ takes integer values $\gamma \in [2,3,4,\ldots\rangle$, where the probability of an observation being lost is $\frac{1}{\gamma}$. When the lost observation control variable is set to 0 then no observations are lost, which is the default setting for most experiments.

The Model

Although the exact dynamics of the environment are known, the model was designed to disregard some of this knowledge, and instead be a simplified representation of the characteristics of the balls. Specifically, the model is unaware of the balls rebounding off the walls, and it has no special logic for dealing with lost observations and noise. Effectively this means that the PQS will create new tracks for balls that have rebounded. This means that the rebounds, noise, and lost observations create an extra level of uncertainty in the model. The model is therefore deliberately imperfect to simulate the fact that the actual dynamics of a system are frequently uncertain. The model is fundamentally the same as the one described in Section 2.5, with the PQML model in Appendix B.2. A few modifications were made as follows:

The model ($\mathcal{M}$) has a track specific section that is unique for each track in the PQS, and the model is designed to follow one ball per track. This track specific section thus holds the current estimated position of the tracked ball and its estimated momentum:

```
float   pos_x    ; estimated position x
float   pos_y    ; same y
float   mom_x    ; estimated momentum x
float   mom_y    ; same y
```

The model is divided into two three separate sections, each reflecting different reasons why the tracker invoked the model:

- *New observation, new track.* A new track was formed with a new observation O_0. The track specific section is reset by assigning *pos_x* and *pos_y* the values of the new observation $O_0.x$ and $O_0.y$, and setting the momentum to zero. The initial score $S_0^{\mathcal{M}}(T_0)$ is set to 0.05.

- *New observation, existing track.* A new observation O_n is assigned to an existing track. Based on the time difference between th last observation in the existing track O_{n-1} and the new observation the current estimated position *pos_x* and *pos_y* is calculated using the stored estimated momentum. The Euclidean distance D between the estimated position (pos_x, pos_y) and observated position $(O_n.x, O_n.y)$ is computed and the new track score $S_n^{\mathcal{M}}(T_n)$ is calculated as:

  $$S_n^{\mathcal{M}}(T_n) = (2 \times S_{n-1}^{\mathcal{M}}(T_{n-1}) + (1 - \frac{D}{0.05}))/3$$

  This basically produces a new score by taking the Euclidean distance between the tracks estimated position and observed position, normalizes it with value 0.05 and subtracts it from 1. So the $(1 - \frac{D}{0.05})$ portion of the score function calculates a normalized score based on the current deviation from the predicted position. This new score is then mixed with the previous track score ($S_{n-1}^{\mathcal{M}}(T_{n-1})$) by a 1:2 weight ratio. Finally, the new momentum is computed by averaging it with the current estimated momentum using a 1:3 weight ratio. (Note that these averaging weight ratios are a very simple way to smooth out noise. A more accurate model, such as the Kalman Filter, would be able to account for the noise directly in the model, leading to improved tracking accuracy. The goal here, however, is to work with a non-optimal model, which is the typical situation for most real-world applications.)

- *No new observation, existing track.* No new observation was added, but the tracker invoked the model anyway. This gives the model the opportunity to update the current estimated position (pos_x, pos_y) based on the stored momentum and the time that has passed since the last time the current estimated position was calculated.

The weighing factors (1:2 for score, 1:3 for momentum) in the second section of the model provide a decent buffer for noise. By smoothing out score and momentum over multiple observations the model is better able to deal with deviations in the input. Finally, the halflife time set for this model was 1 second, meaning that if no new observations are added to a track, then the score of that track decays by 50% each second.

Tracker configuration

Closely related to the specifics of the model are the parameters that are used to configure the tracker. In the experiments some parameters are changed to gauge their effect on the accuracy and efficiency of the tracking system, however, other parameters were fixed to values that make sense, given the model:

- *Track pruning.* The minimum length of a track to be considered for pruning is set to 2, meaning that short tracks are exempt from pruning until they have at least two observations. Additionally, the tracker would only prune tracks that have a score below 0.035.
- *Publication frequency.* Conclusions were published every 0.2 seconds, or 5 times per second. The conclusions are used to evaluate the accuracy of the PQS and the model. The tracker simply takes the current internal hypothesis-set and publishes it to the evaluator program.
- *Observation pruning.* Since the evaluation only considered the last 10 observations in a track (for every time that track was published) all tracks were pruned to hold no more than 20 observations. This simple consideration saves memory and processing time. Additionally, few balls would generate more than 20 observations between rebounds off the walls.
- *Hypothesis size.* Most experiments were done with 10 or fewer balls. To save processing time and memory the tracker was configured to prune as many low scoring tracks as needed to reach a maximum hypothesis size of 20 tracks. Given the environment, this rule, however, was rarely invoked. Setting this value sufficiently large will have no deteriorating effect on accuracy.

The only two parameters that need to be in sync with the model are the minimum length for a track to be considered for pruning (2), and the lowest score at which a track may survive pruning (0.035). Recall that new tracks (length 1) do not have a momentum yet, and therefore their predictions will be off, thus the rule to save very short tracks from pruning allows new tracks to form before their scores are seriously considered. Tracks that do not grow get a score of 0.05 that quickly deteriorates (halflife time of 1 second) to below 0.035 (the tracker pruning threshold) and thus get efficiently pruned away as soon as their length grows to 2 observations or longer.

2.6.2 Performance Metrics

The output of the tracking system gives the last (newest) 10 observations of each track in the hypothesis, 5 times per second. These observations contain a hidden ID label, indicating which ball generated a particular observation. Additionally, the output contains the score that was assigned to a track, the score of the hypothesis, the number of tracks in the hypothesis, and the CPU cycles used by the tracker. The number of tracks indicates if the tracker has correctly determined the number of objects in the environment.

To evaluate accuracy a compounded performance number is used. This compounded performance number reflects serveral factors (all three factors are weighted equally):

- *The length of the tracks.* Basically, longer is better. As tracks grow longer it means that their score was sufficiently high for the tracker to keep them around. Also, if most (or all) of the observations are generated by the same ball, a long track indicates that the ball has been tracked for awhile.

- *The score of the tracks.* An average of the score of all the tracks in the hypothesis. For each track the score is returned directly by the model and indicates the *confidence* that the model has in this track. In other words, the score indicates how certain the model is that all these observations were generated by the same cause. It must be said, however, that lower sampling frequencies have by definition a suppressing effect on the score returned by our model $\mathcal{M}$. So, although the track may be flawlessly following the ball, the score may be lower than at higher sampling frequences, because confidence in the track is lowered by the relatively fewer number of observations per distance travelled by the ball. This number is a measure of the model's confidence in the hypothesis.
- *The deviance in observations.* This number measures the uniformity of the hidden ID labels in the observations of a track. The last 10 observations in a track are evaluated to how many were generated by the same ball. This means that a track with IDs $9,9,9,4,9$ will get a deviance number of $1-\frac{1}{5}$. A track with IDs $3,4,5,6,7,8$ will get a deviance number of $1-\frac{5}{6}$. The first example is a good track following ball with ID 9, and it has one deviant observation from ball 4. The second example is a track made up from observations from 6 different balls, and is therefore following no ball.

Basically, these accuracy numbers are quite abstract and, at lower sampling frequencies may seem worse than they actually are. However, they do give a clearly-stated comparative measure of accuracy that is consistent and considers many different factors. Experience with looking at the actual tracking output in comparison to the compounded accuracy number gives good meaning to these numbers, as described in Table 2.1.

Range	Meaning
20-18	All balls are tracked perfectly. One track per ball.
17-15	Most balls are tracked perfectly. Sometimes a ball is missed, sometimes two tracks get mixed up.
14-10	Tracking goes well for most of the balls. Sometimes everything mixes up and it takes some time to pick up the thread again.
9-6	Typically most balls are correctly tracked, but there are several tracks that contain mixed observations; mixups are more frequent.
≤ 5	Tracking is really horrible. Only a few balls are tracked accurately. Most data is discarded. Tracks are very short due to a lot of confusion.

Table 2.1: Meaning of the compounded accuracy scores used in this performance section.

Evaluating efficiency is more difficult because a PQS is a realtime system. The computational resources used are highly dependent on the specific implementation of the

model, the rate at which observations are arriving, the complexity of the environment, and the computing resources available in the system. Measurements were taken by monitoring the number of CPU cycles used by the application vs. the number of CPU cycles available on the system in the same timeslice. So, for instance, if the PQS tracker used 500,000 cycles in the last n seconds, and the system had 750,000 cycles available, then the load of the tracker would be 67%. Now, if that system had two of these processors, then the available cycles in that same timeframe would be 1,500,000 and the load of the tracker would be only 33%. For this reason all measurements regarding load were taken on the same system.[1] Furthermore, although multiple processors may be available, a single tracker is unable to exploit more than about 1.4 processors at once, due to inherent limits in concurrency using only a single model in the PQS.

2.6.3 Results

This section evaluates accuracy and the efficiency of the tracking of the bouncing balls, both experimentally as well as analytically. This is done by varying parameters such as the complexity of the problem, and the number of hypotheses that the tracker is allowed to maintain. All datapoints in the graphs are averages from at least 3 independent experiments, with 1 outlier removed. Each experiment was done by running the simulation and the tracker for 4 minutes, and measurements reflect the averages of the last 30 seconds of those 4 minutes.

Accuracy vs. Complexity

Intuitively the accuracy of the PQS will suffer from a more complex environment. Factors that make an environment more complex are an increased number of objects, and a reduced sampling frequency. To evaluate the accuracy of the tracker, we experimented with varying sampling frequencies at different ball counts. The graph in Figure 2.11 shows the accuracy of tracking 1 to 10 balls with varying sampling delays. At 0.05 seconds between samples the simulator generated 20 observations per ball per second. Likewise, at 0.5 seconds between samples the simulator generated only 2 observations per ball per second. It is clear from the graph that sampling frequency is much more important than the actual number of balls in the environment. Therefore, as long as there are enough observations flowing in, the tracker is able to keep track of an arbitrary number of objects. (Note, however, that this graph tells us nothing about the computing resources needed, which, intuitively is going to be higher for higher sampling frequencies, and higher for a greater number of balls.)

Accuracy vs. Multiple hypotheses

Contrary to popular belief, increasing the number of hypotheses is not necessarily a way to improve accuracy. It is often said that the number of concurrent hypotheses

[1] A Sun Blade 2500 with dual 1280Mhz SPARC III CPUs.

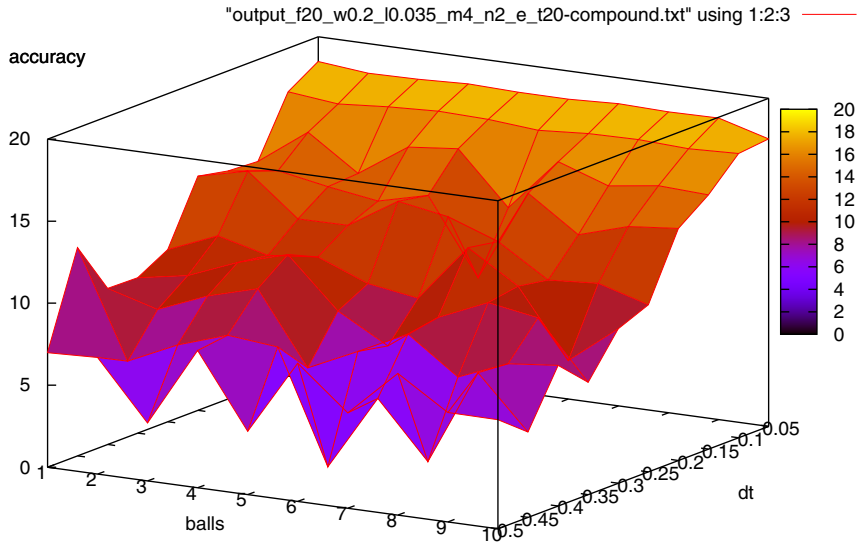


Figure 2.11: Compound performance number for different numbers of bouncing balls and time between samples ranging from 0.05 seconds to 0.5 seconds. After pruning 4 hypotheses are kept.

in a tracking algorithm will allow for more of the initially unlikely combinations to stay around before they can grow to become dominant. However, considering the vast number of observations that flow through the system, the number of possibly generated hypotheses is truly enormous (more about that later). It is true, however, that having *too few* hypotheses impacts accuracy, as the graph in Figure 2.12 shows, where only 1 hypothesis was kept after every pruning step.

Experience in several very different domains has shown that keeping 3 to 4 hypotheses after pruning is sufficient for maximum accuracy in tracking. Experiments with 8 or more hypotheses indicate that accuracy does not necessarily improve at higher hypothesis counts. Considering the PQS algorithm, that makes sense since only the most successful hypotheses contain tracks that have accurately followed a single object for awhile. Therefore, only a few variations on these successful hypotheses need to be kept around to allow for new tracks to form.

It must be said, however, that the optimal number of hypotheses to keep in the system is highly domain dependent. One can conceivably design a multi-model scenario that requires at least 16 hypotheses to be kept, in order to obtain maximum accuracy. In general the rule of thumb is to consider the maximum number of occurring processes

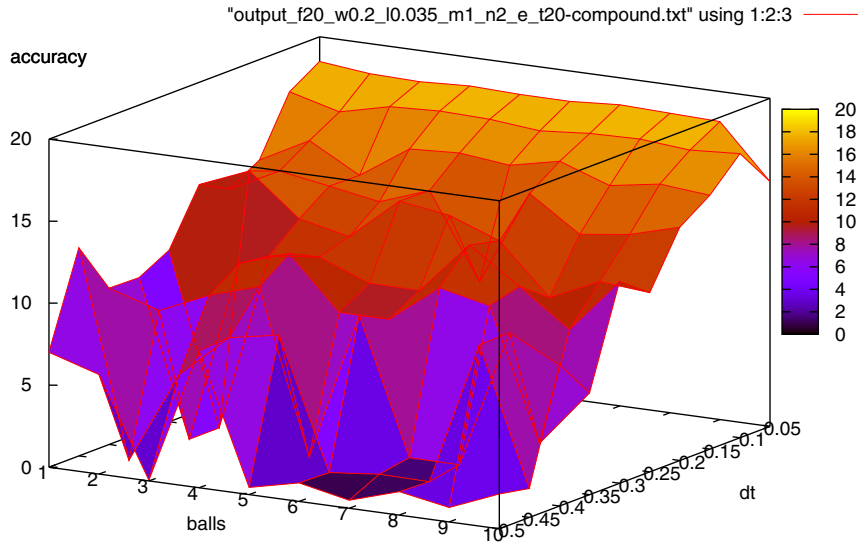


Figure 2.12: Compound performance number for different numbers of bouncing balls and time between samples ranging from 0.05 seconds to 0.5 seconds. Only 1 hypothesis is kept after pruning.

that could easily be confused in the environment. This number should be the maximum number of hypotheses kept. In this case 4 hypotheses is a good number because the chances are very small that more than 4 balls will all be very close together, and travelling in the same direction. This "rule of thumb" is investigated further in the discussion below.

Efficiency vs. Multiple hypotheses

To see the effect of increasing the number of hypotheses on the accuracy and the efficiency we experimented with a rigid simulation setup and changed the number of hypotheses. The graph in Figure 2.13 shows that for different numbers of hypotheses the accuracy of tracking does not change (compound score around 15). However, the total amount of CPU cycles used by the tracker increases linearly, which is to be expected since the amount of work done by the algorithm for each hypothesis in the system is the same. (Note that at higher hypotheses counts the measurement of CPU cycles becomes

less accurate due to OS scheduling and other jobs running on the same system. Number of balls was fixed at 7, with a sampling frequency of 5 observations per second.)

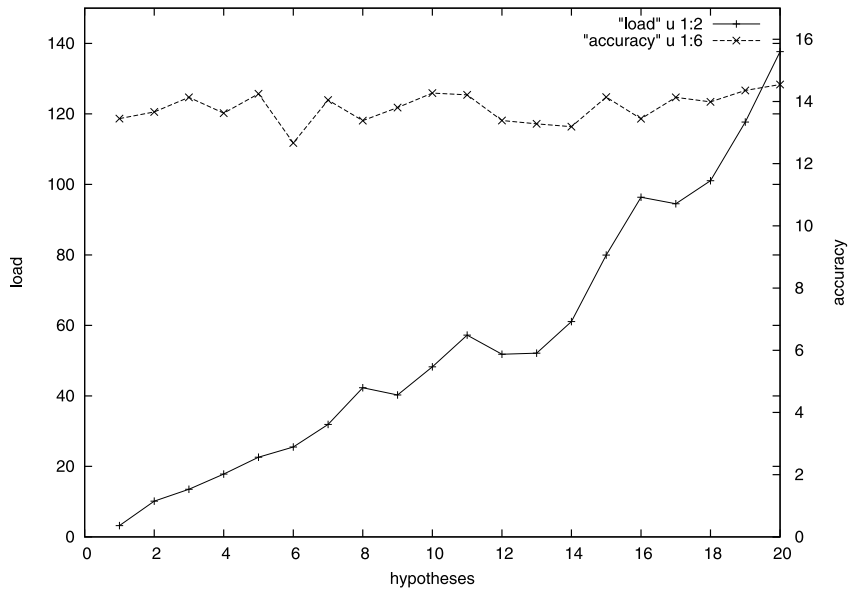


Figure 2.13: Compound performance number as well as tracker load in percentages for different numbers hypotheses. A fixed number of 7 balls was used at a sampling frequency of 5 samples per second ($dt = 0.2$).

Efficiency vs. Complexity

Although the complexity of the environment (ie. the number of balls) was shown to have little impact on the accuracy of tracking, nothing was said about the efficiency at which this can be done. Intuitively the number of combinations that can be made in hypotheses as the number of objects grows is exponential. This means that the amount of work that needs to be done grows exponentially as well, as the number of objects increases. The graph in Figure 2.14 shows the CPU cycles used by the tracker in percentages as the number of balls in the environment grows. Sampling frequency was once again 5 samples per ball per second, and at every pruning step 4 hypotheses were kept. Although the load increases exponentially, it flattens out around 140% where the tracker hits the maximum processing capacity available on the test machine.

This exponential growth can be easily explained when we analyze the rate of hypothesis and track growth if pruning were to be disabled. If a hypothesis has $\mathcal{T}_H(n)$ tracks at time n, then the PQS algorithm expands this to $\mathcal{T}_H(n)$ hypotheses, each with $\mathcal{T}_H(n)$

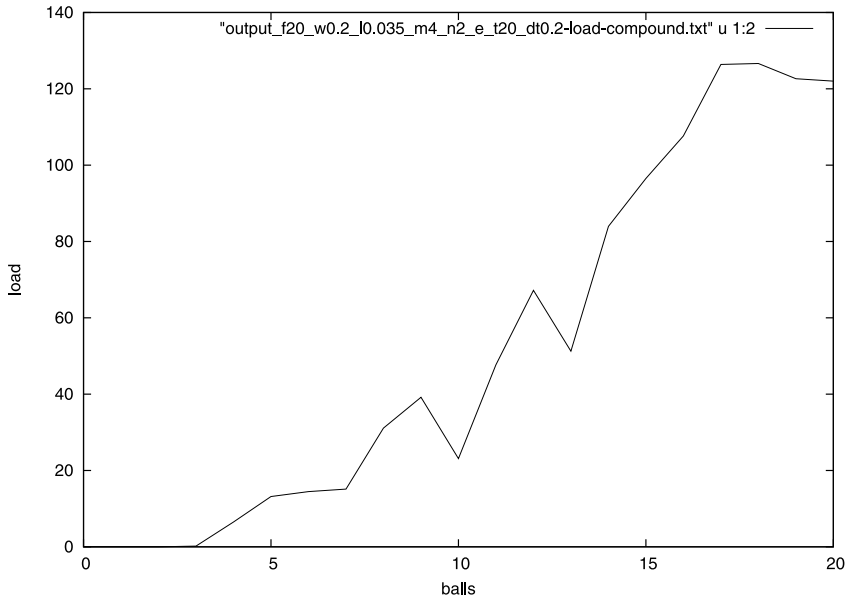


Figure 2.14: The CPU cycles used by the tracker in percentages as the number of balls in the environment is increased. A fixed sampling frequency of 5 samples per second was used, with a maximum of 4 hypotheses.

tracks, and one hypothesis with $\mathcal{T}_H(n)+1$ tracks. The total number of hypotheses that get generated by a hypothesis H when one more observation comes in, given that H contains $\mathcal{T}_H(n)$ tracks is therefore:

$$\mathcal{H}(n+1) = \mathcal{T}_H(n)+1 \tag{2.18}$$

The total number of tracks that hypothesis H will generate when a new observation comes in is:

$$\mathcal{T}(n+1) = (\mathcal{T}_H(n))^2 + \mathcal{T}_H(n)+1 \tag{2.19}$$

The above rules describe the growth of arbitrary hypotheses, where the initial condition is an empty hypothesis-set, which, after the first observation comes in, will hold 1 track in 1 hypothesis. Now, lets consider the total number of hypothesis of size m as $H^m(n)$ at time n. Based on Equations 2.18, and 2.19 each hypothesis with $\mathcal{T}_H(n) = m$ tracks at time n will generate m new hypotheses of size m, and one new hypothesis of size $m+1$ at time $n+1$. This means that the total number of hypothesis of size m at time $n+1$ will therefore be a function of the hypotheses of size m an those of size $m-1$ at time n as follows:

$$H^m(n+1) = H^m(n) \times m + H^{m+1}(n) \tag{2.20}$$

giving the total number of hypotheses at time n as:

$$\sum_m H^m(n) \tag{2.21}$$

and the total number of tracks at time n as:

$$\sum_m H^m(n) \times m \tag{2.22}$$

summing over all hypothesis sizes m. Table 2.2 shows hypothesis and track counts for unrestricted growth where one observation comes in per timestep. When we investigate Equation 2.20 further we realize the following properties:

$$H^0(n) = 1 \quad \forall n \geq 0 \tag{2.23}$$

$$H^m(n) = 0 \quad \forall n < m \tag{2.24}$$

After all, there is always exactly one hypothesis with zero tracks (the empty hypothesis, implying that none of the modeled processes are detected in the environment), leading to any $H^0(n) = 1$. Then, when $n < m$, the count of hypothesis of size m will be zero, which makes sense considering that a hypothesis of any given m size will only produce one hypothesis of size $m+1$, meaning that we need at least n timesteps before we reach a hypothesis of size $m = n$ (this is driven by the second term from Equation 2.20). Now, setting $n = m$ gives:

$$H^m(n) = H^{m-1}(n-1) \quad \forall n = m \tag{2.25}$$

since the first term of Equation 2.20 gets $n < m$, and is therefore zero according to Equation 2.24. Which, when we realize that $H^0(0) = 1$, becomes 1:

$$H^m(n) = H^{m-1}(n-1) = H^0(0) = 1 \quad \forall n = m \tag{2.26}$$

effectively putting all ones on the diagonal $n = m$, and all zeros below it $n < m$. The properties predicted by Equations 2.23, 2.24, and 2.26 can all be observed in Table 2.2. Now we move on to prove exponentiality in the count of all hypotheses of any given size m over time n, when $n \geq m$. By dropping the last term of Equation 2.20 we can infer that at least:

$$H^m(n+1) \geq H^m(n) \times m \quad \forall n \geq m \tag{2.27}$$

expanding the sequence we realize that:

$$H^m(n+1) \geq (H^m(n-1) \times m) \times m \quad \forall n \geq m \tag{2.28}$$

therefore:

$$H^m(n+1) \geq m^{n-m} \quad \forall n \geq m \tag{2.29}$$

thus proving exponential hypothesis counts for all sizes of hypothesis m over time. Finally, thanks to the second term of Equation 2.20, and the property that $H^m(n) = 1 \ \ \forall n = m$ from Equation 2.26, we also know that:

$$H^m(n+1) = m^{n-m} \quad \forall n = m \tag{2.30}$$

$$H^m(n+1) > m^{n-m} \quad \forall n > m \tag{2.31}$$

Hypotheses of size m:	Time n (total observations):									
	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	0	1	4	11	26	57	120	247	502	1013
3	0	0	1	7	32	122	423	1389	4414	13744
4	0	0	0	1	11	76	426	2127	9897	44002
5	0	0	0	0	1	16	156	1206	8157	50682
6	0	0	0	0	0	1	22	288	2934	25761
7	0	0	0	0	0	0	1	29	491	6371
8	0	0	0	0	0	0	0	1	37	787
9	0	0	0	0	0	0	0	0	1	46
10	0	0	0	0	0	0	0	0	0	1
11	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0
Totals:										
Hypotheses	1	3	8	23	75	278	1155	5295	26442	142417
Tracks	1	4	14	51	202	876	4139	21146	115974	678569

Table 2.2: Number of hypotheses of a given size $H^m(n+1) = H^m(n) \times m + H^{m+1}(n)$, the total number of hypotheses $\sum_m H^m(n)$, and the total number of tracks $\sum_m H^m(n) \times m$ for unrestricted hypothesis growth (no pruning).

Now looking at Table 2.2 again we realize what this means. After 10 observations have come in, the total number of possible hypotheses is 142,417, illustrating the complexity of the environment in which PQS operates. With 5 balls in the simulator the first 10

observations come in with 2 samplings, with 10 balls in the simulator only 1 sampling is needed to reach this level of complexity. The graph in Figure 2.15 shows on an exponential scale this growth in hypotheses. Figure 2.16 shows the same for tracks. At each timestep one observation comes in.

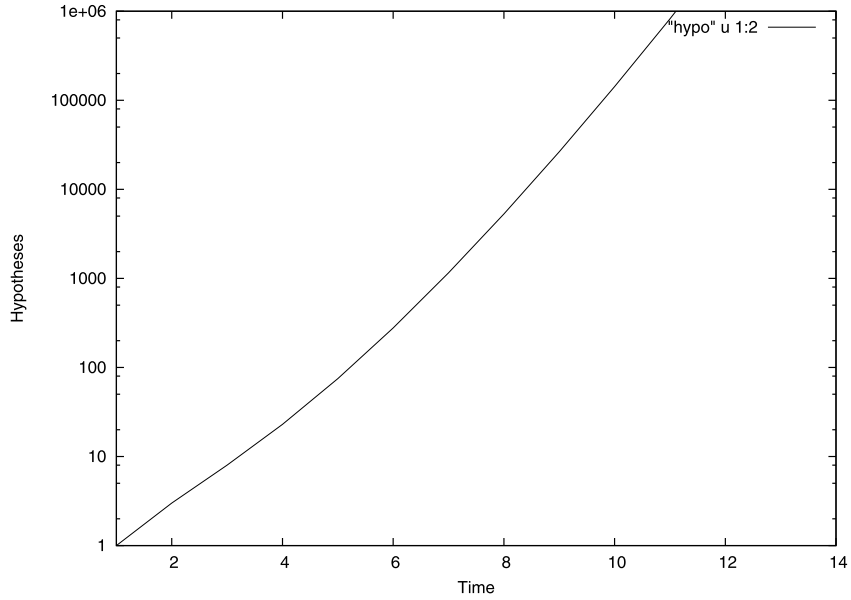


Figure 2.15: Total number of hypotheses in unrestricted growth. At each timestep one observation comes in, so at $t = 10$ the system has received 10 observations.

Noise and Lost Observations

In addition to the time variance and inherent complexity of the environment, we investigated the effect of increased noise and lossy conditions. Most real-world sensor networks suffer from known or unknown inaccuraries in the sensor, with the possibility of lost observations (often the object fails to trigger a sensor, or a sensor broke). The graph in Figure 2.17 shows the effect of a fixed noise level $\eta = 0.002$ and a lost observation rate of $\gamma = 10$. The total number of hypotheses was 4, the number of balls and the sampling rate was variable. Although the noise level was relatively high, serious effects on accuracy are only observed at higher ball numbers or the lower sampling frequencies. This is understandable because the chances of confusion are higher for those conditions. The accuracy was mostly depressed by a dip in the deviance number, meaning that tracks tend to more easily pick up observations that were generated by another ball, thus indicating the increased confusion in the hypotheses.

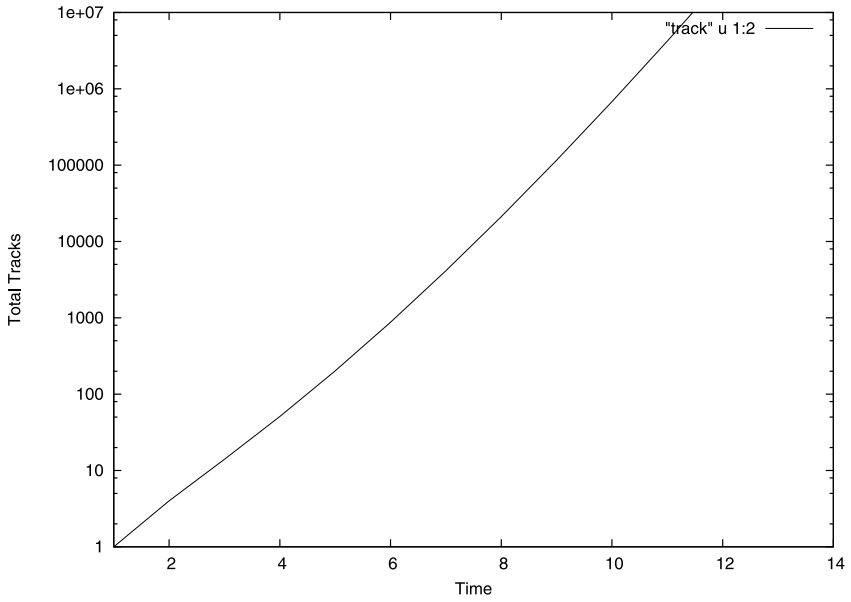


Figure 2.16: Total number of tracks in unrestricted growth.

The graphs in Figure 2.18 and Figure 2.19 respectively show the effect of lost observations and increased noise on accuracy and efficiency, independent of each other. For both series of experiments the number of hypotheses was set to 4, using 7 balls, and a sampling frequency of 5 observations per ball per second. In Figure 2.18 we see clearly that the loss of every other observation $\gamma = 2$ has a suppressing effect on the accuracy of tracking, however, since fewer observations arrive, the required processing time is also less. As more observations come in, the processing time increases, as does the accuracy. This is exactly as one would expect, since there is more information available to correctly track the objects.

Figure 2.19 shows clearly how increased noise can reach dramatically low levels of accuracy. Noise at a level of $\eta = 0.001$ makes it already very difficult for the human eye to track the balls (given 0.2 seconds between observations). As noise further increases not only does the accuracy dip (as expected), however, also the CPU power needed to generate hypotheses increases. Although we would intuitively expect the system to need more cycles to disambiguate the observations when there is more noise, the actual reason for the increase is due to the increased numbers of tracks per hypothesis. Since there was no correction made for noise in the model, the scores that are assigned to the tracks tend to degrade rapidly as more noise is added. This leads to many more smaller tracks being formed, besides the tracks that (more) accurately track the moving balls. Although these

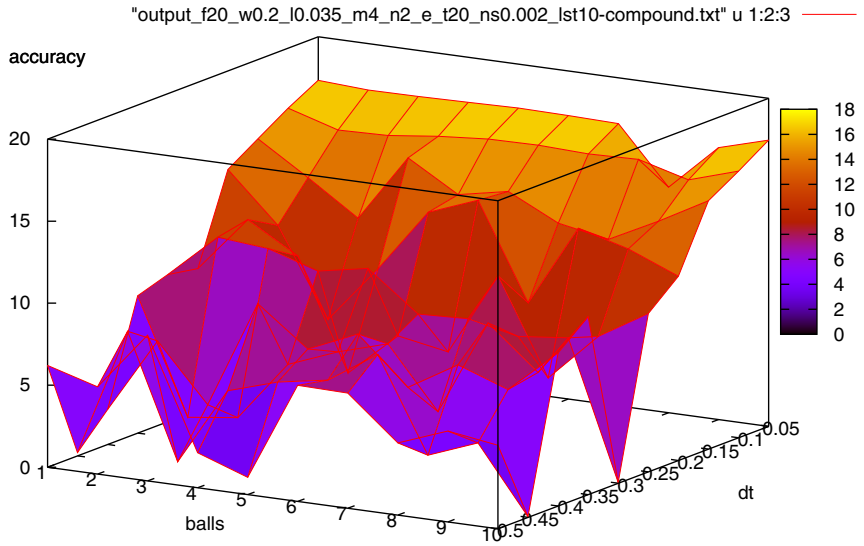


Figure 2.17: Accuracy for a noise level of $\eta = 0.002$ and a lost observation change of one out of $\gamma = 10$. Notice how the graph is significantly more jagged at the lower sampling rates.

tracks are usually pruned away quickly, they do inflict a hit on processing power.

2.6.4 Discussion

Although the results obtained by experimentation do not always correspond to what our intuition tells us, it does correspond with the analysis of the algorithms. For instance, intuitively we would expect tracking accuracy to improve as more hypotheses are kept, however, there is nothing in the algorithm that suggests that keeping more than several hypotheses would be beneficial. After all, the best hypotheses are going to be very similar, and differ only in the minute details, usually concerning small tracks with low scores. Increasing the number of hypotheses does have a linear impact on the total processing time that the tracker uses.

What this means is that there is an upper bound to the number of hypotheses that is going to be useful, meaning that increasing the number of hypotheses past a certain value (in the case of this example: 4) is not going to yield increased accuracy. However, keeping fewer than this upper bound of hypothesis will impact accuracy. This intuitively means that there is an optimal number of hypotheses, for each specific application do-

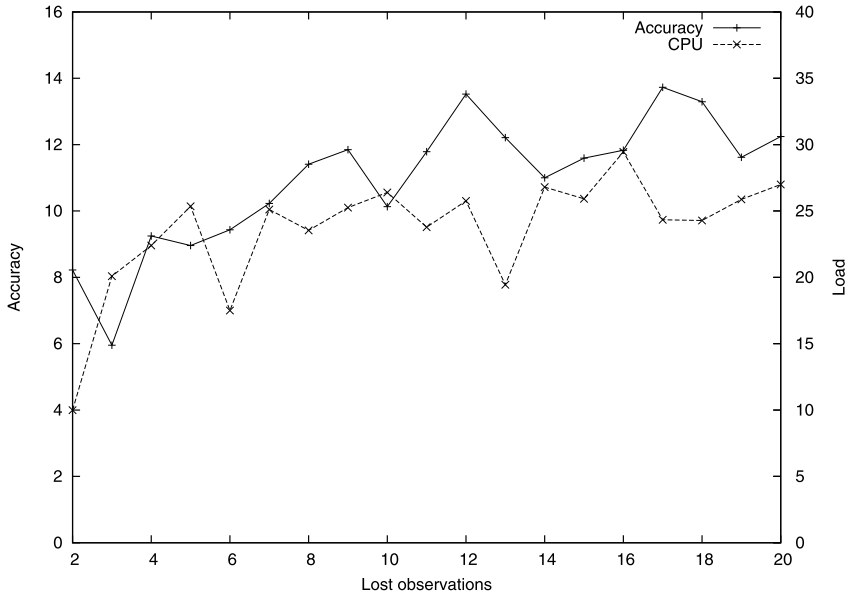


Figure 2.18: As the rate of observations that is lost decreases (horizontal axis, towards the right), the accuracy goes up. As expected, the CPU load also increases, because the PQS has more data to handle.

main. Although it is hard to say what the exact optimal number for each application will be, the general guideline is to consider the maximum number of valid combinations of observations, given the models. Or put more generically: *in the vast majority of the encountered environments, what is the maximum number of processes that can be "easily confused" when occurring simultaneously.* In this case that would mean considering the chance that more than 4 balls are moving very close together at approximately the same speed. Although it may happen, the chances are very small; therefore, using 4 hypotheses is a justifiable number. It must be said, however, that this is a rule of thumb.

Another important fact, although it should not have been a surprise, is that sampling frequency is really the most important factor for accurate tracking. Basically, the more observations that come in, the more information there is available regarding the environment. It only makes sense that sampling frequency is the most important factor in accurate tracking.

Also, analysis of the lower compounded accuracy numbers showed that bad accuracy numbers were mostly due to lower confidence in the tracks, or not correctly tracking every object. In many cases most objects were tracked correctly, but one or two interfering balls would be the cause for many short tracks that all get scored low. This leads to a much lower confidence for the hypothesis as a whole. Additionally, the deviance

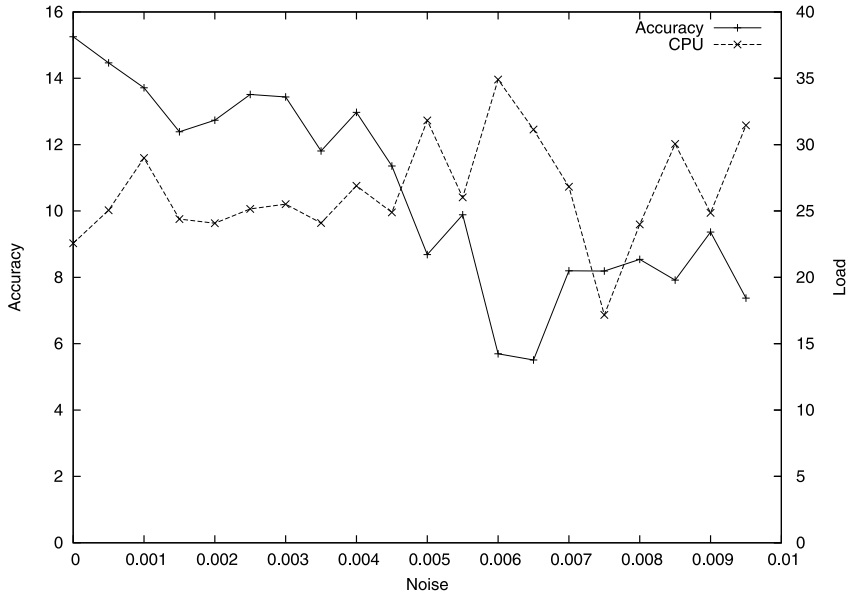


Figure 2.19: As noise increases (horizontal axis, towards the right), the accuracy decreases, and the required CPU time increases. This is expected as the opportunity for confusion gets larger at higher levels of noise.

numbers were rarely below 0.99. This means that longer tracks tend to very accurately disambiguate the observations, even in the most noisy environments.

The fact that the number of hypotheses grows exponentially goes to show how complex the problem is that PQS solves. It also forces a PQS to set strict limits on its use of resources. It is true that the complexity can be reduced if, as with radar, it is known upfront that a set of blips (observations) came from the same sweep. Many combinations do not have to be made anymore, however, in most situations this is simply not the case, as data comes in asynchronously. PQS currently does not offer an easy interface for models to prevent the PQS from forming a worthless hypothesis. Application programmers are therefore forced to make yes/no decisions early on in the model, to improve efficiency. Experiments verified this fact by showing that the number of balls mostly affected load (exponentially), not necessarily accuracy.

Increasing Efficiency

Several things can be done to increase efficiency. Some of those are more structural than others, although efficiency is usually very application dependent. More efficient models, or models that more strongly disambiguate between observations have a strong

effect on increasing efficiency. However, a good structural addition to PQS would be to allow a special model interface to be added where a model can quickly evaluate if a Track-Observation combination is even worth making. In this case the model would not evaluate the track entirely, but be allowed to quickly check whether or not a new observation may be a feasible combination with an existing track. Although mathematical complexity is still exponential, if models implement such a *mayMatch* method efficiently, it can save a lot in processing time.

An example would be for a model to check if the current observation came from the same radar sweep. If that is the case, then the match is probably useless to try. This hypothesis would then never be generated, saving CPU time. Before a hypothesis is created, all models are first asked if a match seems feasible. If any of the models returns "yes", then the hypothesis is actually created and offered for evaluation to the models. This feature is high on the list of future extensions for PQS and the PQML modelling interface.

Increasing Accuracy

It goes without saying that better models will lead to better accuracy, however, it must be said that in the experiments done in this section the model was not adjusted for different sampling frequencies, different levels of noise, or different numbers of lost observations. Most scoring functions would certainly benefit from characteristic knowledge of the sensing infrastructure.

Additionally, it can be read directly from the graphs that a higher sampling frequency and lower observation noise will lead to better tracking accuracy. Finally, it is important to mention that the PQS in this experiment was able to accurately track balls far beyond human capability.

Chapter 3

Case Study: The Spread of Active Worms

3.1 Introduction

Worm detection and the design of the DIB:S system were one of the first indicators that, early on, lead to the development of the Process Query System concept. The idea was to detect the process of a propagating worm by its indicators: aggressively scanning hosts. The input stream would be continous, and most likely massive. Simply put, the question that the system tries to answer is: *given a stream of failed connection attempts, which hosts show clear signs of random scanning behavior.*

Although DIB:S is now a sensor to TRAFEN, it is in itself a Process Query System with just one model: host scanning detection. Using a modern PQS, it is therefore possible to rebuild the DIB:S system in just a matter of hours by constructing a model that processes ICMP destination unreachable observations and looks for evidence of host scanning behavior. In a sense, DIB:S was the first prototype of a Process Query System, with one (built-in) model which is tightly integrated with the data structures. The sensors, then, are the routers that supply the ICMP destination unreachable messages, which are the observations. DIB:S forms tracks by collecting evidence of each scanning host in the network. So for each infected host there will be a track, following the process of that host's scanning behavior. That then is the process model: the increasing stages of more and more aggressive scanning that an infected host goes through as it scans the Internet for other vulnerable systems. DIB:S, however, only keeps one hypothesis, which is the current state of the monitored network (which could potentially be the entire Internet), containing one track for each infected host.

This chapter should therefore be read as an in-depth explanation of the expert knowledge that went into the construction of the DIB:S technology. Although the system was purpose built and has little of the general-purpose features that are typical for an actual Process Query System, the considerations are general and can easily be reapplied to construct a DIB:S model in, for example, PQML. Needless to say, the tight integration of the DIB:S datastructures with the scan-detection model yields a significant performance benefit over using a PQML scan-detection model in a general-purpose PQS. (Specifically, to monitor the complete Internet with a general-purpose PQS it would be required to split up the model in several different tiers, just to be able to cope with hypothesis explosion.)

Although the DIB:S system is in itself a PQS with a hardcoded model, it is used as a sensor for TRAFEN, which was developed as a more general PQS approach. TRAFEN, therefore, should be regarded as the second-tier in the worm detection system; DIB:S tracks the scanning behavior of *individual hosts*, while TRAFEN tracks the evidence of global worms, given the conclusions from DIB:S. In Chapter 4 we will again use the DIB:S conclusions as input to the PQS-Net system, where more advanced models search for evidence of various other types of attacks.

Finally, worm detection and tracking is an interesting case study of the powers of a Process Query System because of the large quantity of infected hosts during a worm infestation. The tier-1 tracking must therefore form tens of thousands of tracks of individual hosts and process their observations. The hypothesis will therefore quickly become very large, considering the massive volume of observations that flow into the first tier (in this specific case: ICMP Destination Unreachable messages). It also shows how a

tier-1 tracker becomes a sensor for a tier-2 tracker, effectively doing data-reduction at every step. The results obtained by the DIB:S/TRAFEN worm detection system make a compelling case for the scalability and versatility of Process Query Systems. A large part of this chapter was published in Chapter 6 of the book: "Managing Cyber Threats" [11].

3.2 Internet Worms

An active Internet worm is malicious software (or malware) that autonomously spreads from host to host, actively searching for vulnerable, uninfected systems. The first such worm was the 1988 Internet worm, which spread through vulnerable Sun 3 and VAX systems starting on November 2, 1988 [111]. This worm exploited flaws in the `sendmail` and `fingerd` code of that time, and through the rsh service and a password-cracking library, also exploited poor password policies. The worm collected the names of target hosts by scanning files, such as `.rhosts` and `.forward`, on the local machine, and then attempted to infect those hosts through the finger, sendmail, and password-guessing exploits. Although the exact number of infected machines is unclear, the worm infected enough machines to disrupt normal Internet activity for several days due to high network traffic and CPU loads.

Recent examples of active worms include Code Red v2, which exploited a flaw in Microsoft's Internet Information Services and infected 360,000 machines [76], and Sapphire/Slammer, which exploited a flaw in Microsoft's SQL Server and infected 75,000 machines [75], in a matter of minutes. Code Red, Sapphire/Slammer and most other recent active worms find vulnerable machines by generating random (or pseudo-random) IP addresses and then probing to see if the desired vulnerable service is running at those addresses. Compared to the 1988 Internet, the modern Internet has so many hosts that random probing is an effective way to find vulnerable machines. In 1988, the address space was sparsely populated, and the 1988 worm, if it had used random probing, would have needed years (or even centuries) to find even one *existing* machine, let alone a vulnerable machine.

In addition to using random probing, most recent worms probe as quickly as possible, so that the worm can spread to most vulnerable machines before system administrators have time to shut down infected machines and repair the exploited security hole. In fact, since current response is entirely manual, a worm only has to spread faster than human response time to succeed. Sapphire/Slammer, the fastest spreading worm to date, far exceeded human response time by infecting most vulnerable machines within five minutes of its launch [75]. Clearly, if the Internet community wants to halt the spread of a worm, rather than simply cleaning up afterward, some form of automated detection and response is needed. Here, we will focus on the problem of *detection*, and present an automated system that can identify active scanning worms soon after they begin to spread. Worm authors, when faced with such a detection system, might switch from address scanning to stealthier techniques for identifying potential targets, including the older, but effective, techniques of the 1988 worm. For this reason, we also will give a brief overview of potential techniques for detecting slow-moving or stealthy worms.

An active scanning worm generates unusual network activity, which can be observed using many possible data sources. One attractive data source is ICMP Destination Unreachable (also known as ICMP Type 3 or ICMP-T3) messages. When a source machine attempts to contact an unreachable or nonexistent target machine, the last Internet router on the path, if configured to do so, will send an ICMP-T3 message to the source machine. Scanning worms, through the process of probing randomly selected IP addresses, will attempt to contact many unreachable machines, and will produce a unique pattern of ICMP-T3 messages. As a worm spreads and infects more machines, more unique source addresses will be attempting to contact the same (or related) ports on unreachable machines. Observing such an increase in scanning activity is a reliable, and early, indicator of worm activity. Using this principle, we have implemented a system called DIB:S in which instrumented routers send copies of their generated ICMP-T3 messages to a central collection station. The instrumented routers are the sensors for the DIB:S system. The collector forms tracks of related ICMP-T3 messages according to the source and destination IP address and ports, and whenever one IP address is seen to be scanning a significant number of target addresses, the collector sends a scan alert to the tier-2 PQS system. This tier-2 PQS system receives the DIB:S scan alerts as observations, each alert containing information regarding one scanning (or scanned) host. TRAFEN, using a higher-level worm model, assembles tracks of related scanning activity, based on the time and target port of the scan, and once a track matches the process model for a worm, TRAFEN takes the track as evidence of an active Internet worm. As we will see, TRAFEN can detect worms before they spread to too many vulnerable machines, opening up the possibility of an effective, early response.

By collecting raw ICMP-T3 messages, the DIB:S system can see scanning activity that spans multiple target networks. Even if a worm instance probes any single network only a few times, DIB:S still will detect the scan as long as there are enough instrumented routers distributed throughout the Internet. The number of instrumented routers can be a modest fraction of the total, making the DIB:S system practical for Internet-wide deployment. In addition, many other unique scanning patterns besides worm propagation can be extracted from the ICMP-T3 data, making DIB:S more extensible and powerful than a system that collects only higher-level scan alerts. Finally, ICMP-T3 messages are relatively compact, and reveal little information about the target network, particularly since the instrumented routers can be configured to send the ICMP-T3 message only to the collection point, rather than to the source machine also. System administrators should have less concern about sharing these messages with a third party than they might with other data sources, although other data sources, as long as they provide insight into scanning activity, could be used within the DIB:S and TRAFEN framework as well.

In the rest of this chapter, we present background on Internet worms and a model for their propagation, describe the architecture of our prototype worm-detection system, DIB:S/TRAFEN, and examine simulation results that illustrate the system's detection performance. Finally, we examine future directions for both worm authors and worm defenders.

3.3 Worms and their Propagation

The first step in detecting an active worm is to understand how active worms propagate, and to develop a general propagation model that can be used as the starting point for detection algorithms. First, we compare active worms with other types of malware, and then we present an epidemic model for worm propagation.

3.3.1 Worms and Viruses

Over the last several years, there has been frequent discussion of the difference between viruses and worms. In the early days after the 1988 Internet worm, Eichin et al. [39] referred to this new event as an "Internet virus", stating that it bore no resemblance to the biological equivalent of a worm. Today, however, most experts refer to it as the "Morris worm", indicating that biological equivalence no longer dictates the terminology. Figure 3.1 is an inheritance graph showing current, commonly accepted relationships in terminology. Viruses and worms are both part of the larger category of malicious code. A related member of the malicious-code group is root-kits and backdoors, pieces of software often installed on compromised systems by hackers to enable them to easily regain control of the machine in the future. Rootkits are associated with the so-called "auto-rooters", pieces of software that offer a nice GUI to the hacker, making computer intrusion child's play. A disturbing detail is that many of these tools can perform multiple attacks (exploits) with various target selection strategies, eliminating the need for any understanding from the hacker. The tools often are easier to use than most security products.

Another related member of the malicious-code family is spyware, software that ships and installs with bona fide programs and relays information from the user's computer back to a data center without the user's explicit consent. This implies that the user often is not aware that spyware programs are present on the system, increasing the risk that private, or even privileged, information might be stolen. Spyware is gaining more attention lately, largely because software packages are increasing in size and complexity, making detection of spyware much more difficult. In addition, spyware programs tend to remain on the system even when the program to which they were originally attached is removed.

Where other malicious code is intended for controlled use, viruses and worms are designed to propagate without control. This makes them very dangerous, since there are no bounds on their spread, and their workings are fully decentralized. Where root-kits and backdoors provide the hacker with full control of a system, worms and viruses need to be fully autonomous, following the same algorithm over and over again for each newly infected system. There is no reason, however, why the two cannot be combined. creating a massively (self-)propagating piece of malware that leaves backdoors for the hacker to enter all infected systems at will. Regarding terminology, worms and viruses can be viewed as separate types of autonomous malware (as we prefer and depict in Figure 3.1), or viruses can be viewed as a broad category of which worms are a special case. Whether worms are their own category or a subcategory has little effect on the discussion of their properties, so we leave it to the reader to form their own opinion.

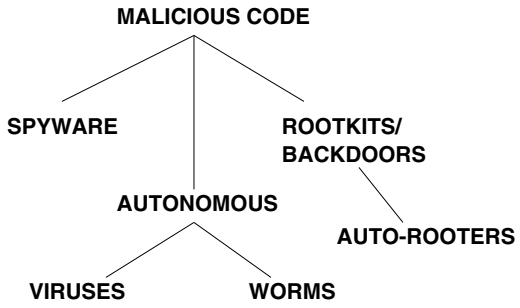


Figure 3.1: A partial hierarchy of malicious code (or malware).

The difference between worms and viruses lies in their method of propagation. In short, viruses require carriers, where worms facilitate their own propagation. Worms often use an attack strategy that actively selects targets and opens connections to those targets. The worm then launches an exploit, and, if successful, propagates by copying its code to the new system and then running that code. The new system now is infected and will behave the same as the system that infected it, resulting in two copies of the worm, both looking for new systems to infect. This spread continues until most vulnerable systems are infected, or until a built-in timer stops the propagation and switches the worm to another mode, such as a massive Distributed Denial of Service (DDOS) attack using all the infected systems as drones.

In contrast to worms, viruses need a carrier to propagate. Traditionally, viruses bind to executable files, the system boot sector, or both. This ensures that the virus is loaded into memory at boot time, or whenever a program is loaded. Once active in memory, the virus binds to the operating system and tries to infect the boot sector and every program that is run. This will guarantee its spread, since infected executables that are run on clean systems will infect the boot-sector of that system, leading to subsequent infection of that system's other programs as well. This technique requires executable files to be shared between computers, imposing a natural limit on how fast the virus can spread. Recently, however, viruses have been designed to piggy-back on bona fide communication mechanisms such as email. Email viruses often rely on the recipient to open the email and run the attached viral executable, which, in turn, will attempt to send itself to all e-mail addresses in the user's address book. This is the reason that such viruses often come from your best friend. Virus writers use many techniques to hide the actual virus from the user, such as embedding the viral code inside a screensaver or game. A more sophisticated approach is to include a macro in the e-mail that will run the viral code as soon as the e-mail is opened (without the user having to open the viral attachment itself). This approach, however, requires an email client that understands and automatically interprets and runs such macros. Email viruses, with or without automatic execution of the viral attachment, show propagation patterns very similar to those of active worms.

3.3.2 Worm Spread

The propagation pattern and autonomous behavior that classifies worms leads to a clearly identifiable three-step algorithm:

1. Target Selection
2. Infection Attempt
3. Code propagation (when the infection attempt succeeds)

Intuitively, the faster a worm can identify and infect new vulnerable targets, the faster it can propagate. This is important, since historically it seems that slow and "silent" worms do significantly worse than fast and "loud" worms, in terms of the peak number of infected systems. The major reason for the success of fast worms is the minimal response time that they allow to take appropriate countermeasures. Successful response mainly depends on human factors, since it usually involves system administrators learning about new worm events, and then identifying and patching or removing any vulnerable systems in their networks. Given the limits of human response time, the initial propagation of a new worm can proceed unobstructed, giving fast worms the chance to reach a "critical mass", namely, infect enough systems to create and sustain an epidemic. In the next section, we will back these intuitive explanations with some basic epidemiology.

The target-selection algorithm is crucial to the success of a worm, and worm authors have shown stunning creativity in this part. Proposed or observed approaches include (1) random (directed or hitlist), (2) sniffing, and (3) name (email addresses, system files, DNS). In addition, many worms have combined these three techniques with varying results. The most common, and easily implemented, algorithm is random generation of target IP addresses. This method has gained popularity on the IPv4 Internet, since the IPv4 Internet is densely populated. Selecting a random IP address has a high chance (between 5% to 35%) of hitting an existing machine. A larger address space, like IPv6, would mitigate this problem since it would take *years* to even find an active IP address by random scanning. On the IPv6 Internet, the random IP generation technique is far less effective due to the enormous address space. When all the machines currently on the IPv4 Internet migrate to the IPv6 Internet, the IPv6 Internet would be extremely sparse, and it would take *years* for a random-generation algorithm to find an IP address actually associated with a host.

To improve the chance of finding vulnerable machines, many worm authors employ techniques in which they direct the random target selection. By preferring address ranges that are densely populated or address ranges that are suspected to contain a large number of *vulnerable* machines, the worm can propagate significantly faster. As an example of the latter case, the vulnerability that the worm exploits might be typical of home computers. The worm author would attempt to identify up front which target ranges hold the most home computers (dial-up, DSL, and cable-modem ISPs) and then program the worm to prefer targets in those address ranges. Alternatively, the worm can be programmed to select targets only from a list of *known* targets. This approach usually is called "hitlist propagation", and is most effectively used as an initial propagation method

before defaulting to random propagation [114]. Such a hitlist would contain IP addresses that are known to be vulnerable systems, and thus would need to be constructed before the worm was released. Construction of hitlists can be done slowly over the course of months by randomly scanning the Internet. To avoid attacking the same system multiple times during propagation, the list can be split in half every time a worm instance propagates. One half is kept by the infecting system, while the other half is given to the newly infected system. Hitlists are an effective way of establishing a critical mass of infected systems. A further optimization is permutation scanning in which every worm copy scans according to the same reproducible random sequence. Each copy of the worm starts at a position in the sequence determined by the IP address of the local host, and switches to a new random position whenever it encounters a machine that already is infected [114]. Switching to a new position takes into account the fact that if a host is already infected, some other copy of the worm already is working its way through that portion of the sequence. The permutation approach, which has not been employed in actual worms yet, preserves the simplicity of random scanning, while minimizing duplicative scanning effort.

Scanning activity can be difficult to hide, since traffic-monitoring and intrusion-detection systems can notice the pattern of one machine actively connecting to many other machines. A technique that has been frequently discussed, although not used in implemented worms yet, is passively sniffing the network (or inspecting application-level traffic) to identify reachable IP addresses that likely are running a service that the worm can exploit. As an example, a contagion worm might have two exploits, one for web clients and one for web servers. A copy of the worm on a web server attempts to infect any web client that requests a page, while a copy of the worm on a web client attempts to infect any web server to which the client connects. Fortunately, this approach is applicable only for some services, since the worm must see enough traffic to build up a reasonably sized set of potential targets. For example, if the worm only had an exploit for web servers and was passively sniffing the network to identify other web servers, it might see little or no traffic for any web server other than the one already infected, particularly given the prevalence of switched Ethernet. On the other hand, a worm exploiting a vulnerability in email servers will have a better chance of succeeding, since email servers contact *each other* to exchange email. As long as users on the local network make moderate to heavy use of email, the worm will be able to identify a significant number of email servers that it can attempt to infect. As an added bonus for the worm author, such an email worm would be equally successful in densely or sparsely populated address spaces.

When the address space is only sparsely populated, random scanning (even to construct a hitlist) can be an impossible task, and thus other methods need to be employed. In addition to the passive network sniffing discussed above, a worm can use DNS names rather than IP addresses to identify systems. When a top-level domain name is acquired, DNS servers often will reveal the names of the associated mail exchange server and Web server. Even if these names are not obtainable directly from the DNS system, the worm author can make an educated guess as to what the names of existing systems would be. Imagine that the worm acquired the domain *exampledomain.com*. A logical nam-

ing scheme would suggest that *www.exampledomain.com* would be the Web server and *mail.exampledomain.com* would be the mailserver. A list of other names would include *www1, ns, ns1, dns, dns1, nameserver, ftp, smtp, pop3 or skywalker*. Names from Greek mythology also are very popular. The worm author's creativity can be endless, and techniques that have been used for many years in password crackers also can be used to construct hostnames. If a site has a hostname *sparc09*, for example, it is worth trying *sparc01, sparc02, ... sparc99* as well. Additionally, hostnames can be gleaned from many other sources. The 1988 worm [39] used the *.rhosts* file to obtain hostnames of other systems in the network. Similarly, most operating systems maintain small name databases as a backup for when the DNS system fails. Other sources can be email addresses, which have the basic structure *username@domainname*, and provide domain names for the process above. Obtaining the addresses or names of potential targets from information stored on the currently infected machine often is called topological scanning [114]. Although most worms today use some form of random target selection, the introduction of IPv6 means that it is no longer the guaranteed fastest way to propagate. Future worms most likely will employ combinations of the above techniques to facilitate their propagation. In addition, viruses that use normal network traffic as a carrier will become increasingly popular, since they do not need to select their own targets.

After a target is selected, the worm will attempt to infect it. If successful, the worm will run a copy of itself on the newly compromised system. The two general approaches to code propagation are the use of a central repository or the use of cloning. A central repository stores the code of the worm, and each time a new host is infected, the worm is downloaded from this location. The benefits of this approach are that the worm author can update the worm code while the worm is propagating. It is even possible to store commands that each instance of the worm will download and execute, leaving all the infected systems open for control by the worm author. The drawback is obvious – the security community easily can take the central repository off-line, effectively disabling the worm and stopping further propagation. The second approach (cloning) does not allow for code updates, but also makes it much harder to stop the worm during propagation. The code of the running worm is copied and started on each newly infected machine, effectively cloning the current copy. Although not allowing any control mechanisms, these types of worms tend to be more successful. Evolutions of the central-repository technique, or programming worm copies to create their own peer-to-peer network for command distribution, will provide significant control capabilities for hard-to-stop worms, however.

3.3.3 Epidemics

To get a feel for the factors that govern worm (as well as virus) propagation, most researchers take to the classic epidemiological equations. These models describe biological epidemics quite well, and have proven to be very applicable to their cyber equivalents. We will introduce these models here and refer to further reading for a more in-depth coverage of the topic.

In its most basic form, the behavior of a single host is described by the SIR (Susceptible - Infective - Recovered) model as shown in Figure 3.2. For a given worm, the *S*-state

(susceptible) indicates the host is vulnerable to that worm. The *I*-state (infective) indicates that the host is infected and spreading the worm. The *R*-state (recovered/removed) means that the host is not (or no longer) of interest to the epidemic. The reasons for being in the *R*-state may vary, most often the host simply was not vulnerable to the worm in the first place, or the host was patched (whether infected or not). Alternatively, the host might be disconnected from the network, either to prevent infection or further propagation. For any worm, only a marginal portion of all the hosts are vulnerable, i.e., in the group of susceptibles *S*. The majority of Internet-connected hosts will be in the *R*-group, and not be involved in the spread of the epidemic. The transitions between the states are given below, keeping in mind that the transitions apply to the state of a host for one particular infection only:

S → I (infection)
I → R (patching or disconnection)

And furthermore:

S → R (uninfected system patched)
I → S (infection removed, but system not patched)
R → S (susceptible system reconnected to the network)
R → I (infected system reconnected to the network)

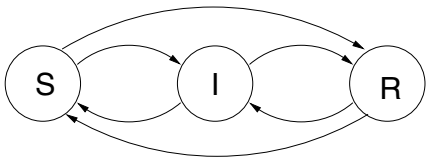


Figure 3.2: The SIR (Susceptibles-Infectives-Recovered) model is probably the most popular way of identifying the groups in an epidemic, and its transitions form the basis for a broad range of mathematical models.

The first two transitions are the most common case, and account for the majority of the total number of state transitions made during an epidemic. They model the infection of vulnerable systems (S→I transition), and the patching or removal of infected systems (S→R transition). Many systems generally are not vulnerable to a certain worm attack, and such systems do not change state and largely remain in their *R*-group. The classic epidemic equations from Kermack and McKendrick focus on these two transitions (see Daley and Gani [32]) for modeling the spread of an infection in continuous time. The population N is constructed from the three groups S, I, and R, which change over time as defined by $s(t)$, $i(t)$, and $r(t)$, where t_0 is the time at which the infection begins. Note that $N = s(t) + i(t) + r(t)$, meaning that the population size is assumed constant, which is acceptable considering that we defined R to contain disconnected, not just patched, systems. The population changes over time can be defined as

$$(a) \quad \frac{\mathrm{d}s}{\mathrm{d}t} = -\beta si \qquad (b) \quad \frac{\mathrm{d}i}{\mathrm{d}t} = \beta si - \gamma i \qquad (c) \quad \frac{\mathrm{d}r}{\mathrm{d}t} = \gamma i \tag{3.1}$$

The parameter β models the transition S→I and γ models the transition I→R. Intuitively, β is the likelihood of one particular infected system contacting (and infecting) one particular susceptible system in dt time. Likewise, γ is the likelihood that one particular infected system is patched or disconnected in dt time. Putting a number to these factors is not easy since it is different for each worm. The general principels discussed in the previous section, however, lead to some guidelines. First, the rate at which a worm can infect new systems is limited by the rate at which it can contact other systems (which determines β). This rate is either limited by parallelism or by bandwidth, whichever reaches its limit first, and these factors are determined by the capabilities of the infected host and the target-selection algorithm of the worm. The most effective propagation would be when the worm uses up all the bandwidth that the host has to offer, thus, the closer a worm can approach this limit, the better its chances are for fast propagation. There are several factors involved that make this easier or harder. The first factor is the protocol that the worm uses to propagate. When the worm uses a fire and forget protocol (like UDP), it most easily can use all of the bandwidth since it never has to wait for a return packet. When a connection-oriented protocol (such as TCP) is used, however, the worm will need to wait for an acknowledgment from the target host before it can send the attack data. The choice is not always up to the worm author since most services (and hence most vulnerabilities) are built using connection-oriented protocols.

The latency between initiation and acknowledgment, however, can be filled with connection requests to other potential targets when the worm interleaves them properly. With appropriate programming, which may include the worm generating its own connection requests and bypassing the operating system's network stack, the worm can hide most of the latency associated with connection-oriented protocols. For example, one thread in the worm would craft requests packets, transmit those packets, and log the outstanding connection in a table, while a second thread constantly would check (sniff) for return packets and attempt to match them with the entries in the table. Every several seconds the worm traverses the entire table to fault connection requests that have not had a response within a *worm-defined* timeout period. By making this table sufficiently large, the worm should be able to fill the available bandwidth without needing to run thousands of concurrent copies of itself on the infected host. Although such an approach makes the worm more complex and more difficult to implement correctly, the added burden might be well worth the increased propagation speed. In addition, matters become easier when the target-selection algorithm has some predetermined knowledge of reachability or even vulnerability, such as in the case of a pre-constructed hitlist. The targets on this list are mostly reachable and likely also vulnerable. This will significantly decrease the time spent in timeouts, meaning that more network bandwidth will be used since each connection attempt will take less time on average.

Given this discussion, the average time that each connection takes can be calculated as:

$$\tau = r \times t_{latency} + (1-r) \times t_{timeout} \tag{3.2}$$

where r is the reachability based on the target-selection algorithm. A perfect hitlist would give $r = 1$, and random target selection on the current Internet would give $r \approx 0.1$. (It must be noted that general reachability on the Internet is affected not only by assigned and used address space, but also by firewalls and other filtering devices. Additionally, reachability numbers vary majorly between address ranges. Safe numbers for r are usually between 1% and 25%, depending on the service under attack, preferred address ranges, etc.)

When a worm does use a hitlist for initial propagation, the worm would have two different values for β, one value for the hitlist part of the propagation, and a second smaller value for the remaining (random) part of the propagation. In addition, I_0 (the initial number of infected systems) for the second part would be the number of infected systems after the hitlist propagation is complete. For completely random target selection, β can be defined as

$$\beta = \frac{1}{N} \times \frac{\alpha}{\tau} \tag{3.3}$$

where N is the size of the address space (2^{32} in case of IPv4) and α is the number of concurrent scanning threads. In the case of a worm that implemented a fully parallel scan through the construction of its own request packets, α might be defined quite high (even if the worm itself only used the two threads described above). In the equations, dt is the same as the unit of τ, meaning that if τ is calculated in seconds, dt in Equations 3.1 also is in seconds. For a perfect hitlist (where every IP address is indeed a susceptible host), we instead could define β as:

$$\beta = \frac{1}{S_0} \times \frac{\alpha}{\tau} \tag{3.4}$$

where S_0 is the number of systems that are initially susceptible (assuming that the hitlist holds *all* susceptible systems). The second factor $\frac{\alpha}{\tau}$ essentially calculates the average number of successful connections a single infected host can complete in dt time (not all of those are necessarily susceptibles). When network bandwidth is the limiting factor, rather than worm parallelism, the second factor can be replaced with a division of the available network bandwidth by the size of the infection packetstream.

The γ parameter (the I→R transition) can be harder to model since it mostly depends on actions of the system administrator. (See Figure 3.3, where parameter β was calculated based on Equations 3.2 and 3.3 and the characteristics of Code Red v2 on a per-second basis: Code Red v2 used 100 concurrent scanning threads ($\alpha = 100$), with an average reachability of $r = \frac{1}{10}$ and a default timeout on no response of 21 seconds (based on the default Windows NT timeout, exponential back-off with 3 retries after 3, 6, and 12 seconds). This gives (3.2) $\tau = \frac{1}{10} \times 1 + (1 - \frac{1}{10}) \times 21 = 19$. The address space was IPv4, and thus $N = 2^{32}$ gives (3.3) $\beta = \frac{1}{2^{32}} \times \frac{100}{19} = 1.23 \times 10^{-9}$. The term $\frac{\alpha}{\tau} = \frac{100}{19}$ gives us about 5.26 completed scans per second. Notice how the total number of systems infected (surface area under the graphs) decreases with higher values for γ. Code Red v2 data was collected at TRIUMF Canada (http://www.triumf.ca), which generously made the data available to us for this research.) It will take security personnel some time to discover a

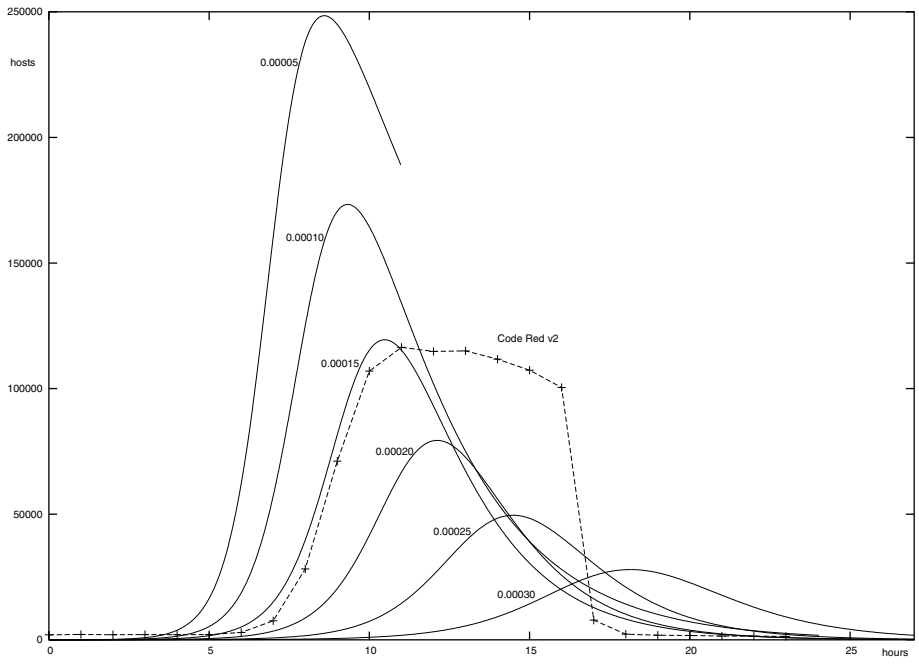


Figure 3.3: Spread of Code Red v2 versus the epidemic equations for different values of γ. The vertical axis represents the total number of infected systems at any given time, and the horizontal axis is the time in hours.

newly launched worm, and then they will need to analyze the worm and possibly write a patch. System administrators then must learn about, download, and install the new patch. Another option for system administrators is the disconnection of infected machines from the network. Both processes, patching and disconnection, are hard to model, and likely are not governed by a fixed rate. Note that in the Kermack and McKendrick model, the transition is dependent only on the current size of the group of infected systems, which could be too simple a dependency to model the behavior of security personnel and system administrators.

Additional Transitions. The other transitions in the SIR model are interesting for further study. The S→R transition models uninfected systems that are vulnerable to the worm under consideration, but get patched or disconnected before they are infected. Although this process might be underway before the worm is launched (i.e., a patch for the worm's exploit is available a priori), only its effect on the worm should be modeled. Patching that occurs before the worm is released simply decreases S_0. Below is an extended set of differential equations taking into account all six transitions from the graph:

$$\frac{ds}{dt} = -\beta si - \eta i + \zeta r + \theta i \tag{3.5}$$

$$\frac{di}{dt} = \beta si - \gamma i + \varepsilon r - \theta i \tag{3.6}$$

$$\frac{dr}{dt} = \gamma i - \varepsilon r - \zeta r + \eta i \tag{3.7}$$

The S→R transition is governed by the parameter η and is taken to be dependent on the size of the group of infectives over time. This is parallel to the I→R transition, and indicates that administrators will patch uninfected systems, as well as infected ones, with greater urgency as the worm propagates. It can be argued, however, that it should be multiplied by the size of the group of susceptibles as well, since the chance that administrators patch or disconnect uninfected systems decreases as there are fewer systems uninfected. The I→S transition (represented by θ) also is taken to increase and decrease as the group of infected systems grows and shrinks. This means that the larger the group of infected systems, the greater the number of systems that will be cleaned, but not patched. A good example of this was the propagation of the Code Red v2 worm. Although Code Red v2 could be removed from a system by rebooting, the system would be susceptible to re-infection after the reboot. The R→S transition (ζ in the equations) is most likely due to uninfected, yet susceptible, systems being taken off-line and then reconnected to the network later. It also could indicate systems that were patched and updated, but became vulnerable again for various reasons. In nearly all cases, this will be a small fraction, and is mostly dependent on how many systems are disconnected or patched. Similarly, The R→I transition (modeled by ε) will be small, representing the infected systems that are taken off-line and later reconnected, allowing them to continue spreading the infection. One final note on these four transitions is their relative insignificance compared to β and γ. Even for very small values of θ, ε and ζ, the equations can be unrealistically imbalanced. The interested reader is encouraged to try different values for all parameters and see how the epidemic curve behaves.

3.4 Response

The best way to respond to an epidemic is to prevent it in the first place. History has shown, however, that there have always been unpatched vulnerabilities. Moreover, with software getting more and more complex, it is unlikely that this will change. Software vendors put significant effort into distributing patches to mend security holes in their software, but not nearly enough users install such patches promptly. Often people are not aware of security updates, and many others get tired of the continuous stream of updates, inadvertently leading to disregard. Patching does decrease the size of the pool of susceptibles, however, effectively limiting the damage any worm can do. The most obvious solution would seem to be automated patching services, although the necessary basis of trust is lacking. Most operating systems vendors offer some form of automated patching

service, either manually invoked, or automatically, and such services work reasonably well. The catch is, however, that these services often only patch the vulnerabilities in the base operating system and several frequently used network services. They have no way of knowing about custom installed services, which may subsequently remain vulnerable to attack. Also, security experts have long argued that these services are themselves an important target for attack, although none have been reported compromised so far. It can be seen, however, that if an exploited automated patching service goes undetected, the attackers have the option of creating a large pool of susceptible systems by distributing a vulnerable patch. Needless to say, such a situation would be devastating.

Thus, although automated patching does have its place, automated response after the worm is launched must be a critical part of an effective defense. When we consider the epidemic equations, the two parameters that govern the majority of all transitions are β and γ. An epidemic can be reduced by either lowering β or increasing γ. Figure 3.3 shows how increasing the value of γ reduces the number of hosts affected by the epidemic (surface area under the graph). We now will discuss several ways of influencing these parameters as a form of active response to worms.

Increasing γ. A common way of avoiding communication with infected systems is the 'blacklist'. This is a technique often used within the security community to filter out IP addresses that have shown aggressive behavior in the recent past. A similar technique could be used to collect IP addresses of systems that are known infectives. This list would grow as the worms propagate. Routers and firewalls across the world would have to implement filtering rules to disallow traffic from any of these IP addresses. This effectively cuts infected systems off the network by blocking them from communicating, therefore increasing γ. The R-group will increase, and there will be relatively more disconnected, infected systems than normal. Problems with this approach are the implementation requirements. Moore et al. [77] conclude that practically all of the Internet's major connections need to employ blacklist filters for this technique to be effective. In addition, the list of blocked IP addresses needs to be continually updated and, as the list grows, it will incur a significant load on all the participating routers and firewalls. Additionally, a fast and accurate detection system needs to be in place to determine which systems should be added to the blacklist. Another common problem that this technique poses is the ability for attackers to perform a DOS attack on arbitrary hosts or networks. Attackers can spoof malicious traffic, making it seem like it came from a particular network, and get the worm response system to blacklist or filter out all traffic from that network, effectively disconnecting it from the Internet.

Reducing β. Since β governs the growth of the worm, worm authors will try to maximize β to speed up the propagation; the security community, in turn, must try to minimize it. A technique that has been discussed by Williamson [124] is to reduce the number of new connections that a host may initiate per timeslice. A connection is counted as new when it connects to an IP address that it was not communicating with in the recent past. Known IP addresses (i.e., those with which a machine communications often, such as mail or DNS servers) are stored in a list of a given size and will never incur a delay. For the unknown IP addresses, however, the connection limit is imposed incrementally. A worm, which created a list of hundreds of IP addresses to contact, would incur a delay

between itself and the previous connection request. The connection limit is suggested at five new connections per second, which is roughly the effective scanning speed of the Code Red v2 worm, meaning that only the fastest of worms will be hindered.[1] An additional argument for implementing this method is the minimal overhead it puts on the system, while putting a direct limit to β. Some server systems, however, would suffer badly from this method, since they usually have more active outbound connections. Consider, for example, DNS or email servers, both of which will connect to many other systems based on the name queries or email messages sent by the users. A similar difficulty is encountered on multi-user systems, where multiple users are logged on at the same time. This technique works better on "static" servers like webservers that mainly listen for incoming connections. Additionally, it may be possible for a worm to circumvent the rate-limiting mechanisms by crafting packets instead of traversing the TCP/IP stack.

A second technique is "traffic content filtering". It is based on the idea that routers and/or firewalls will test all traffic flowing through against a set of known, viral signatures. When a malicious signature is detected, the packet is dropped, effectively limiting the propagation of malicious code and decreasing β. The technique, however, requires very elaborate signatures and matching on port/protocol combinations, since the sheer volume of traffic traveling through large routers creates a fair possibility that smaller signatures would be matched in regular, bona fide traffic. As Moore et al. discuss [77], for application during a new worm event, this approach requires the signature to be generated as early as possible. Signature-capable routers would need to be in widespread use, as well as a mechanism to quickly and securely distribute new signatures. Once again, this defense system allows for a DOS attack when an attacker is able to insert a falsified signature that would block all traffic for a particular service. In addition, this system would put a tremendous overhead on critical network routers on the Internet, since signature matching (especially when the pool of signatures is large) is very processor intensive. Combined with the need to re-assemble each fragmented packet, to avoid overlooking fragmented attacks, this cure might be difficult to deploy widely.

Conclusion. The general mantra for this section is the need for very early detection of new worm events. Whatever the response will be, it will never be useful if the alert and classification come too late. Considering that the Sapphire/Slammer worm [75, 38] propagated in just several minutes, it is clearly not humanly possible to generate the alerts by hand. Although automated alert and response systems would be up to the task, they are at the risk of becoming the target themselves, potentially being more dangerous than any regular worm could ever be. It seems, therefore, that there will always remain a delicate balance between human interaction and machine automation. We can envision a system in which the monitoring and detection is done automatically, such that alerts and signatures are generated for a human first responder to assess. Next the human responder can decide which (if any at all) of the active response mechanisms to activate, allowing an appropriate response to the event.

[1] This number was previously calculated as $\frac{\alpha}{\tau} = \frac{100}{19} \approx 5.26$, as well as observed from the actual worm in our test environment.

3.5 Early Detection of Scanning Worms

Our prototype system for detecting scanning worms collects ICMP Destination Unreachable (or ICMP-T3) messages as observations from instrumented routers (the sensors), then aggregates these messages in tracks to identify scanning activity, and then looks for patterns of scanning activity that indicate a propagating worm using a tier-2 PQS. The system, whose architecture is shown in Figure 3.4, has two major components, the Dartmouth ICMP BCC: System or DIB:S (tier-1), which aggregates the ICMP-T3 messages into scans alerts, and our Tracking and Fusion Engine or TRAFEN (tier-2), which identifies propagating worms based on their scanning activity. DIB:S forms tracks of individual host activity, while TRAFEN generates tracks of related scanning activity based on the submitted worm process models. These process models contain several approaches to identifying worm-like behavior, for instance, by searching for a near-exponential rise in DIB:S alerts. From a layered PQS point of view, the DIB:S system is the tier-1 tracker and forms a *sensor* to the tier-2 TRAFEN system, however, from the worm detection point of view, DIB:S and TRAFEN should be considered integrated components. (Technically, as mentioned before, a general-purpose PQS system could be used to perform the specific-purpose task done by DIB:S, however, this would come at a significant performance cost.) The prototype DIB:S and TRAFEN system focuses on the detection of fast-moving worms that attempt to infect the vulnerable population within minutes or hours, since automatic detection of such worms is the only way to provide enough early warning to take appropriate countermeasures. For slower-moving worms, manual detection of the worm, in time to warn system administrators and other personnel, is much more likely. As we will see, however, the system can detect of slow-moving worms through straightforward extensions. The system will not detect worms that do not probe "randomly" generated IP addresses, however, such as worms that use a pre-constructed hitlist or that monitor network traffic to identify existent, reachable machines. In this section, we present background on ICMP-T3 messages, describe the DIB:S and TRAFEN components, and examine the detection capabilities of the prototype system.

3.5.1 ICMP-T3 Messages and Instrumented Routers

When a source machine attempts to contact a nonexistent or unreachable machine, an Internet router, somewhere between the source machine and the target network, will determine that the packets can go no farther. This router, if configured to do so, will send an ICMP-T3 message to the source machine. Scanning worms, through the process of probing randomly selected IP addresses, will attempt to contact many unreachable or nonexistent machines, such as machines protected by a firewall or addresses from an unassigned part of the Internet. If this scanning activity produces enough ICMP-T3 messages, we can infer the presence of a propagating worm through its unique scanning pattern, specifically, the growth in scanning activity as the worm infects more and more machines.

Table 3.1 shows the responses we received when we probed selected address ranges on the Internet. The data, which was obtained for a separate project, is skewed slightly,

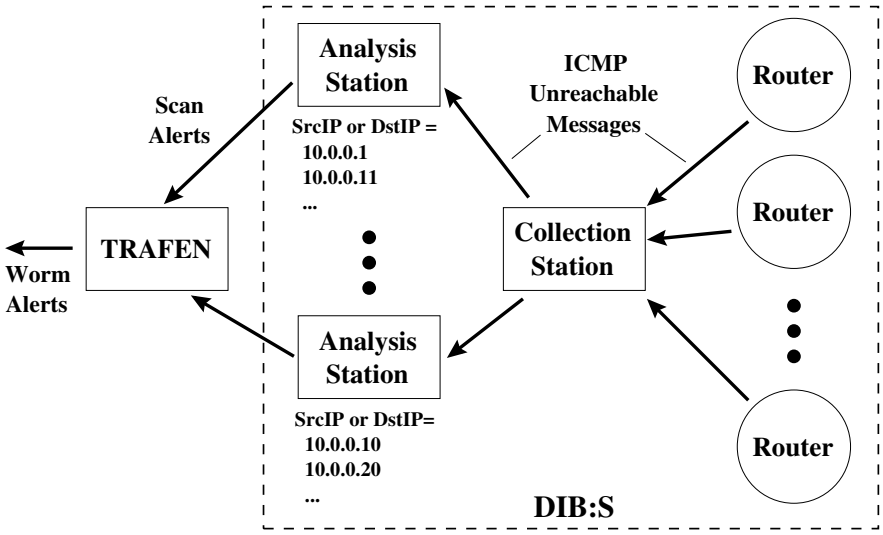


Figure 3.4: The combined DIB:S and TRAFEN system. ICMP Unreachable messages with the same source or destination address are sent to the *same* analysis station.

since we scanned only known highly populated address ranges (such as DSL-pool and cable model ISPs). Many address ranges simply are unassigned, and contain no reachable machines at all. The two most significant numbers are the high response rates (25% average) and the numbers of ICMP-T3 messages returned (6.2% average). The latter number, although seemingly low, means that a significant fraction of scan attempts will produce an ICMP-T3 message at some router. Thus, if we can collect and analyze ICMP-T3 messages from multiple, distributed routers, we will have enough messages to detect a worm's unique scanning activity.

	PING 24.[0-128]/16		PING [209-211].[32-64]/16		TCP/80 [209-211].[32-64]/16	
Requests	1628977	100%	6487973	100%	1171298	100%
No response	1258388	77.3%	4911425	75.7%	800636	68.4%
Echo replies	244445	15.0%	636135	9.8%	37707	3.2%
ICMP-T3	77361	4.7%	398841	6.0%	104555	8.9%
Other	48783	3.0%	550472	8.5%	228400	19.5%

Table 3.1: Responses to random probing on the Internet - ICMP echo request on the 24.0/16 - 24.128/16 networks. ICMP echo and TCP port 80 request on the 209.32-64/16 - 211.32-64/16 networks

Due to privacy concerns, we have chosen *not* to sniff for ICMP-T3 messages, but instead to ask network providers and other organizations to forward the ICMP-T3 messages

from their routers to our analysis systems. These forwarded messages are essentially a Blind Carbon Copy (BCC) of the original ICMP-T3 message, which is a legitimate action since the generating router was a participant in the original conversation. Although site policy may require that no response be sent to the source machine, the router can remain silent to the outside world while still sending the ICMP-T3 messages to the analysis systems. In particular, there was no response to 75% of our probes, but many of these probes may have gone through routers that were instructed to silently ignore unsolicited traffic. These routers could easily forward ICMP-T3s to the analysis systems, while still dropping the original packet without a response to the sender. [2] This approach allows broader coverage, while still respecting the security policies of individual organizations. We currently provide router patches for the LINUX kernel to provide the ICMP-T3 forwarding ability.

ICMP-T3 messages come in several different flavors, [97] two of which are of particular interest for detecting scanning activity: Network Unreachable (Code 0) and Host Unreachable (Code 1). A router generates a Network Unreachable message when a desired network cannot be reached. This might happen when a packet is sent to an IP address that resides in an unassigned portion of the Internet address space. Far more commonly, a router generates a Host Unreachable message when a router cannot find the addressed host in its network. This might happen when the packet could be routed to the correct network, but the router responsible for that network could not locate a machine in its network that bears the requested IP address.

The feature that makes analyzing ICMP-T3 messages useful is their message body. When a router builds a Destination Unreachable message, it includes the IP header, and at least the first eight bytes of the body of the *original message* (i.e., the message that provoked the ICMP-T3 response) as the payload of the ICMP-T3 message. (See Figure 3.5.) For TCP and UDP, this includes the source and destination port numbers. Scanning systems thus will reveal both their IP address and their target port.

To clarify this, imagine a system that is randomly scanning IP addresses to find web servers listening on Port 80. From Table 3.1, we see that this will elicit a significant number of ICMP-T3 responses that would include, as their payload, the IP address of the scanning system, plus at least eight extra bytes from the original payload, which, in this example, would be the beginning of the TCP header. If the analysis systems received a BCC of such an ICMP-T3 packet, the IP address of the scanning system would be known, as well as the port for which it was scanning.[3] Now, if multiple routers across the Internet forward the ICMP-T3s that they generate to the data analysis systems, it would soon become clear that a host (i.e., an IP address) was scanning the Internet to find web servers. We refer to this scanning pattern as a "bloom".

[2] RFC 1812 section 5.2.7.1 states that routers *should be able* to generate ICMP-T3s, not that they *should* generate them.

[3] The destination port appears within the first 8 bytes of the TCP or UDP header.

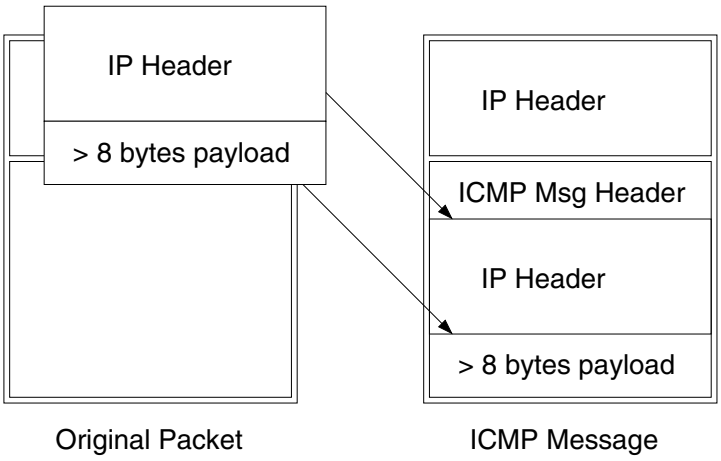


Figure 3.5: The IP header, plus at least 8 bytes of the protocol data, of the packet that provoked that generation of an ICMP-T3 get copied as the payload of that ICMP-T3 message.

3.5.2 DIB:S

The primary task of DIB:S is to collect ICMP-T3 data and identify blooms of scanning activity. The instrumented routers, described in the previous section, send carbon copies of their ICMP-T3 messages to one or more collectors, which, in turn, will forward the messages to one or more analyzers. Each analyzer is assigned an IP address range within which it will look for scanning activity, and more analyzers can be spawned dynamically as needed (with appropriate updates to the assigned address ranges). Each analyser will therefore only form tracks for a certain range of IP addresses, thus balancing the load. When an ICMP-T3 message arrives at a collector, the collector extracts the embedded content, sends one copy of the message to the analyzer associated with the embedded source IP address, and sends another copy to the analyzer associated with the destination IP address.[4] In this way, an analyzer will see all information about a specific range of IP addresses, regardless of the routers from which the information came. Organizing the analysis by source and destination address, rather than the generating router, is critical, since randomly scanning worms will hit many different networks, and the resulting ICMP-T3 messages will come from many different routers. This then, is at the core of the embedded process model, tracks collect all evidence pertaining to just one host, regardless of where the information originated. Thus, the scanning activity is much more visible when viewed across routers, rather than at a single router. This division of labor is possible since tracks are formed containing evidence of one infected host per track. Tracks can therefore be distributed deterministically.

[4]Depending on the number of analyzers and the particular source and destination IP addresses, the two copies might go to the same analyzer, in which case only one copy is actually sent.

The analyzers keep a history of the ICMP-T3 messages received for a particular IP address over the last Δt seconds. DIB:S will generate alerts in six cases. Only two are relevant to worm detection – in the last Δt seconds, on the same port p and using the same protocol P, one host has contacted N different IP addresses (Case 1), or one host has been contacted by N different IP addresses (Case 2). These are classical scanning patterns, both observed during worm propagation, although Case 2 also can indicate a failed server for which requests keep arriving. Similarly, the other two cases also are symmetrical – in the last Δt seconds, on the same port p and using the same protocol P, one IP address has contacted another IP address at least N times (Case 3), or one IP address has been contacted by another IP address at least N times (Case 4). These two cases could indicate service failure In addition to the four primary cases, DIB:S also can generate alerts in two symmetrical secondary cases – in the last Δt seconds, one IP address has contacted another IP address on at least N different ports (Case 5), or one IP address was contacted by another IP address on at least N different ports (Case 6). The DIB:S alerts contain the case number, the embedded source and destination IP address, the protocol, and, if available, the source and destination port numbers. An analyzer will not generate a second alert for an address if it already has generated an alert within the last Δt seconds. As time progresses and the history expires, however, the analyzer will generate a second alert if scanning activity from or against the address continues. If one IP address is scanning two different ports, DIB:S will issue two separate alerts.

The proper values for the parameters N and Δt are the configuration of the tier-1 model, and depend on the number of participating routers, however, several general things can be said. A lower value of N increases the chances of false positives, and any value below $N = 4$ makes the system unusable. Although higher values will lead to more accurate detection, the moment of detection will be later, possibly *too* late. Experimentation has shown that $5 \leq N \leq 15$ gives the best results. Similarly, smaller values for Δt will give a very inaccurate view of events, since alerts on fast scanning IP addresses will be frequently re-issued, and slower-scanning worms will not be detected at all. Higher values of Δt, however, put a serious performance penalty on the analysis system since each packet has to be remembered for a longer time. Proper values during experimentation were determined to be $300 \leq \Delta t \leq 14400$. We will consider these two parameters in more detail in a later section.

Case 1 is the most direct and obvious indication of (random) scanning behavior, and also is the clearest case of a "bloom". It could be a direct result from an active worm. Case 2 would be seen when a public server fails and requests keep arriving. If Case 2 increases over time instead of decreasing, it is most likely the result of a successful denial-of-service (DOS) attack. Cases 3 and 4 are most likely the result of one of two communicating hosts going off-line, and packets from the other system are attempting, but failing, to reach the first host. Cases 5 and 6 could be a sign of one system doing a vertical port scan on another system. It is very unlikely that DIB:S will see these cases, however, since ICMP is rate limited, and routers generally will not generate more than three or four ICMP messages er second. Thus, it takes significant time to see these latter types of observations. In fact, a worm simulation with a steady stream of injected noise showed that 99% of all observations are actually of Type 1.

3.5.3 TRAFEN model

This section describes the initial elementary models that were submitted to TRAFEN (TRacking And Fusion ENgine), to track active worms. Later, in the *PQS-Net* setting (see Chapter 4), we experimented with more diverse sensor inputs and more complicated models for worm tracking. It should be noted, however, that the general worm model described below performs as well, if not better, than these more complicated PQS-Net worm models.

Recall that a Process Query System can take input from arbitrary sensors and then forms hypotheses regarding the observed environment, based on the process queries given by the user. *Process Queries* are synonymous with *Process Models* or simply *Models*. To apply a PQS to a particular problem domain, the developer must (1) define the format for the input observations, and (2) must provide one or more process models that return a score representing how related a set of observations in a track are. From the tier-2 point of view, in which we detect the actual worms, the observations are the scan alerts from the DIB:S analyzers, and the score assigned to a track is the probability that the track represents a worm. TRAFEN subscribes to the DIB:S Alert stream and picks out the Case 1 and Case 2 alerts (since those are the most relevant for worm detection).

The score calculation, then, is the heart of our model, and is essentially a set of scoring rules. This is different, however, from a rule-based detector which looks for specific signatures. Keep in mind that the models are invoked at the arrival of each observation, therefore the scoring rules will essentially "grow" and "shrink" the score of the tracks. Thus the model focusses on how well new incoming observations fit in an existing track. The model assigns two *match* scores, one representing the scan *type* (destination port), and the other representing the *timescale* between two subsequent scans of the same type. After initial experiments, we arrived at three straightforward rules. *Rule 1 (type):* If a machine scans the same port, using the same protocol, as the machines already in a particular track, the type match is *high* (0.9); otherwise the type match is *low* (0.1). This rule captures the fact that an active worm typically scans for and exploits one particular vulnerable service, although the rule could be extended easily to take into account those worms that scan two or more *related* service ports. *Rule 2 (time):* If a machine performs a scan only a short period of time after a previous series of scans, the time match should be higher than if the scans occur farther apart, which captures the fact that an active worm must scan continuously if it wants to propagate quickly. We assign a time match of 1.0 if the new scan occurs 10 seconds or less after a previous scan, a time match of 0.0 if a new scan occurs 300 seconds or more after a previous scan, and a time match scaled linearly between 0 and 1 if the scan is between 10 and 300 seconds after the previous scan. Although the exact thresholds have little effect on tracking performance, these thresholds are best for fast-moving worms. *Rule 3 (combined):* Finally, if the type match is low, the overall score (the score returned by the model) representing that the new scan is related to the tracked scans is set low, again 0.1, because two scans on different destination ports likely do not represent the same active worm, no matter how closely together those two scans occur in time. If the type match is high, the overall score is set between 0.675 and 0.925, scaled linearly according to the time match. Since

the score of an initial single-observation track is set to a low value, and since the *track* score is a moving average of these individual scores, the rules ensure that it takes several observations for the track probability to increase significantly, reflecting the fact that only a series of scans can indicate a worm.

Finally, a more complex worm detector could provide two process descriptions to TRAFEN, one for worm activity and one for coincidental scanning activity, allowing TRAFEN to create hypotheses in which each track has a *type*, namely, *worm* or *coincidental scan*, and greatly reducing the possibility of false positives. Overall, the TRAFEN framework allowed us to produce a working worm detector (given the DIB:S input) in only a few hours, and provides the flexibility to extend the tracking system later through more complex models. Next, we will examine the detection performance of the current ruleset, and discuss extensions to the current DIB:S/TRAFEN system.

3.6 Performance

Evaluating the performance of a multi-tiered system must focus on the performance of the system as a whole, rather than evaluating its parts. In this case that intuitively means *how quickly does the system detect new worms?* Needless to say, performance measurement of this system is dependent on the number and placing of the routeres (i.e. the sensors), the configuration of DIB:S (i.e. Tier-1), and the model(s) used in TRAFEN (i.e. Tier-2). This section discusses the various environments and parameters sets with which the systems performance was evaluated.

3.6.1 Simulating Worms

DIB:S and TRAFEN currently are deployed at Dartmouth College, with instrumented routers throughout the USA sending their ICMP-T3 messages to our DIB:S installation. This initial deployment, although functional, is not enough to analyze the detection performance of the system because there is no way to accurately determine *ground truth*; what is actually happening. Therefore we turn to simulated worms for performance analysis. We developed two different worm simulations, one small-scale and one large-scale. The small-scale simulation allows us to run hundreds of worms through the DIB:S/TRAFEN system in rapid succession, allowing us to explore the parameter space and fine-tune the system for specific environments. The large-scale simulation is essentially the same, but it simulates a worm propagating over the *entire* Internet, allowing system evaluation under more realistic conditions. The volume of ICMP-T3 messages generated in the large-scale simulation can be massive, however.

Small-Scale Worm Simulation Our small-scale worm simulator is designed to run worms on address spaces of one million addresses or less. The number of reachable hosts and the number of susceptible hosts is configurable, and each susceptible host is simulated individually. We assume that each reachable system is reachable from all connected hosts, using a given latency distribution, and we do not explicitly simulate routers. Instead, the generation of ICMP-T3 messages is done based on address ranges.

For example, when the router coverage is set to 10%, ICMP-T3 messages are generated for a fixed 10% of the addresses (and only for those addresses within the 10% that do not correspond to a reachable host). For a random address probe, the simulation first checks whether the address is associated with a vulnerable host, then whether it is associated with a reachable host, and finally, if not reachable, whether the address is covered by an instrumented router. When the probe hits a vulnerable host, the worm propagates to that host, and the newly infected host starts scanning as well. In our experiments, typical network parameters are a space of $10^5 - 10^6$ addresses of which 5-15% are reachable and 100-1000 hosts are vulnerable. The only worm-specific parameter is the worm's scan rate, and the worm selects random target addresses uniformly distributed through the address space, with the random seed for each worm instance derived from the current (simulated) time and the address of the infected machine.

Large-Scale Worm Simulation

The large-scale worm simulator aims to be an accurate representation of the current Internet. The address space contains 2^{32} addresses and is subdivided into *Autonomous Systems* between which simulated BGP-routers route traffic. The simulation is divided into two tiers, the macroscopic level and the microscopic (or network) level. The BGP-routers are simulated at the macroscopic level, where a stochastic version of the epidemic model is used to model the total flow of infection packets *between* autonomous systems. At this level, only the size of the flow and the source of the flow (a distribution of autonomous systems) is simulated. Then, for several representative (1-128) autonomous systems, the actual networks and the infected, susceptible, and reachable hosts are simulated at the microscopic or packet level. Because necessary address information is missing the generated traffic is an estimation. The arrival of packets is modelled as a Poisson process; the interarrival times between packets shrink as the flow of traffic increases. The ICMP-T3 messages are generated at the border of participating autonomous systems, under the assumption that those autonomous systems are connected by a single gateway. The ICMP-T3 forwarding routers only look at arriving scan packets, sending ICMP-T3 messages to a real DIB:S/TRAFEN system when a scan hits an IP address that was not represented by an actual host. The generation of ICMP-T3 messages is rate limited at 3 per second per router, the same limit applied by Linux and CISCO routers. An in-depth description of this simulation system can be found in [69].

3.6.2 Detection Capabilities

Small-Scale Worm Simulation. Figure 3.6 shows the detection performance of DIB:S and TRAFEN for a simulated Sapphire/Slammer worm. The y-axis is the percentage of vulnerable machines that are infected at the time of worm detection, and the x-axis is the router coverage. Each line in the graph corresponds to a different network size. For each network size, 75% of the addresses were unreachable, 25% of the addresses were reachable, and 0.1% of the addresses were reachable *and* vulnerable. For example, for a network size of 500,000 unique addresses, 375,000 addresses are unreachable, 125,000 are reachable, and 500 are vulnerable. The reachable 25% corresponds to our observed data from the scans of selected *populated* address ranges, while the vulnerable 0.1%,

although large, corresponds to a vulnerability in web, mail, database, or other widely installed software. Each data point in the graphs is an average across ten simulated worms, and each simulated worm probed 100 target addresses per infected machine per second, slightly lower than, but consistent with, the average Sapphire/Slammer scan rate. DIB:S had to receive $N = 5$ ICMP-T3 messages for the same IP address before issuing a scan alert to TRAFEN, and DIB:S maintained a history window of $\Delta t = 300$ seconds. Each simulation run continued until the worm infected all vulnerable machines, and TRAFEN was assumed to have detected the worm as soon as the probability of a track containing the relevant scanning activity went above a likelihood threshold of 0.9, a constant value used in all experiments. The value of 0.9 is arbitrary and based mostly on experience. Generally, if the track-score in TRAFEN is "low" then there is no worm, and when it is "high" then a worm has been detected.

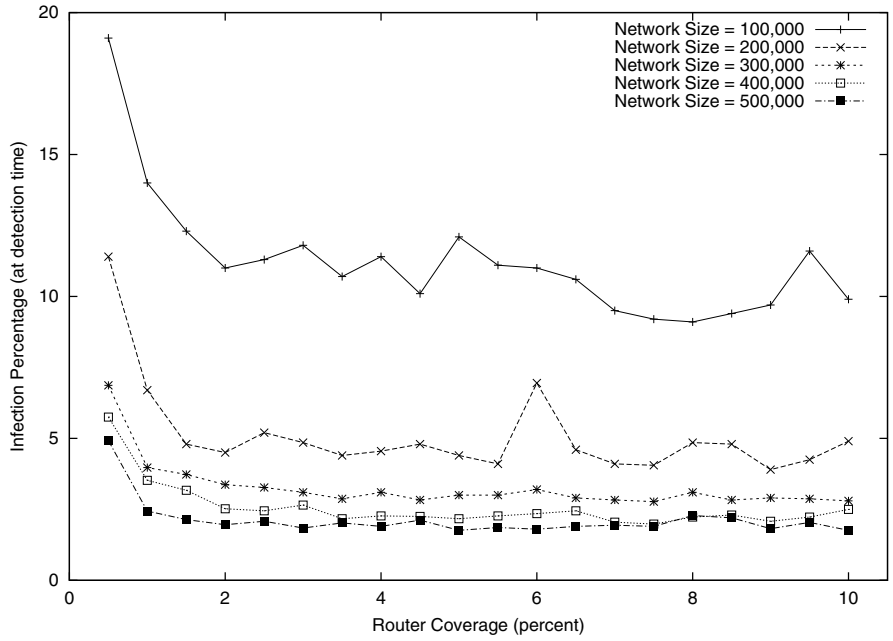


Figure 3.6: Detection performance with the small-scale simulation. The horizontal axis is the router coverage, and the vertical axis is the percentage of vulnerable machines already infected at the time that an active worm is detected.

As seen in Figure 3.6, the detection performance improves significantly as the router coverage increases from 1% to 2%, but then levels off at different, roughly constant, values for the different network sizes. For a network size of 500,000, for example, the infection percentage starts at a peak of 5% when the router coverage is 0.5, but drops quickly to around 2% as the coverage increases. The straightforward reason is that, for router coverages of 2% and higher, DIB:S receives enough ICMP-T3 messages to

reliably detect the scanning activity of the *first* few infected machines. Thus, at these higher coverages, the detection always will take place within a fixed number of infected machines, no matter whether the coverage is 2% or 10%. For router coverages below 2%, however, DIB:S will not receive enough ICMP-T3 messages to reliably detect all scanning activity, and correspondingly more machines will be infected before DIB:S can conclude that a worm is present. The critical message of this graph is that router coverage of 2% provides just as good detection performance as higher coverages, meaning that we need only a modest number of instrumented routers, and that we need only transmit and process a manageable volume of ICMP-T3 messages.

In addition, the detection performance improves as the network size increases. Figure 3.7 shows how many systems were infected at the time of detection, for several different simulated Internet sizes. The worm was identical in all of the cases, and spread with the same parameters. The number of vulnerable hosts was grown proportionally with the size of the simulated Internet. The explanation is simply that DIB:S detection performance is dependent not so much on the percentage of machines infected so far, but on the absolute number of infected machines and the amount of scanning activity that the worm generated while infecting those machines. In fact, if the infection percentages for a 2% router coverage are converted into the absolute number of infected machines, we can see that detection occurs at approximately 12 infected machines in a 100,000-address network, and at a comparable 10 machines in a 500,000-address network. Intiutively we can say that, assuming the error in the sensor data is low, once a dozen or so machines have been infected, showing an exponential increase in number of infected systems, we will detect a worm. However, if this exponential increase is lacking, the detection will not happen. In our simulations this exponential increase is usually very distinct, however, in the case of an actual worm the initial increase in infected systems may not be distinct enough for a solid detection at a dozen or so infected hosts. This can be due to a relatively low sensor (router) coverage compared to the worms propagation speed. Overall, in terms of our ability to detect the worm early and eventually protect the largest *percentage* of vulnerable, but not yet infected, machines, we can keep the router coverage fixed, and still do better and better as the network size increases. Alternatively, for a larger network, we can achieve the same detection performance with a smaller router coverage.

Large-Scale Simulation. The large-scale simulation allows us to explore these network-size results further. The large-scale simulation used 2^{32} addresses, and instrumented routers were placed at the border of class-B sized networks. Each of those class-B networks were assumed to have 50% unused address space, and each router was rate limited at 3 ICMP-T3 messages per second. Two worms were simulated for router coverages varying from 1 class-B participating router up to 64 class-B participating routers. The first worm, a simulated version of Code Red v2, scanned at a rate of 5.65 scans per second with a population of 380,000 susceptible hosts, and the second worm, a simulated version of Sapphire/Slammer scanned at a rate of 4000 scans per second with a population of 120,000 susceptible hosts. The DIB:S parameters were $N = 5$ and $\Delta t = 7200$ for the Code Red v2 worm, and $N = 5$ and $\Delta t = 3600$ for the Slammer/Sapphire worm. The higher values of Δt are necessary since the number of instrumented routers is small

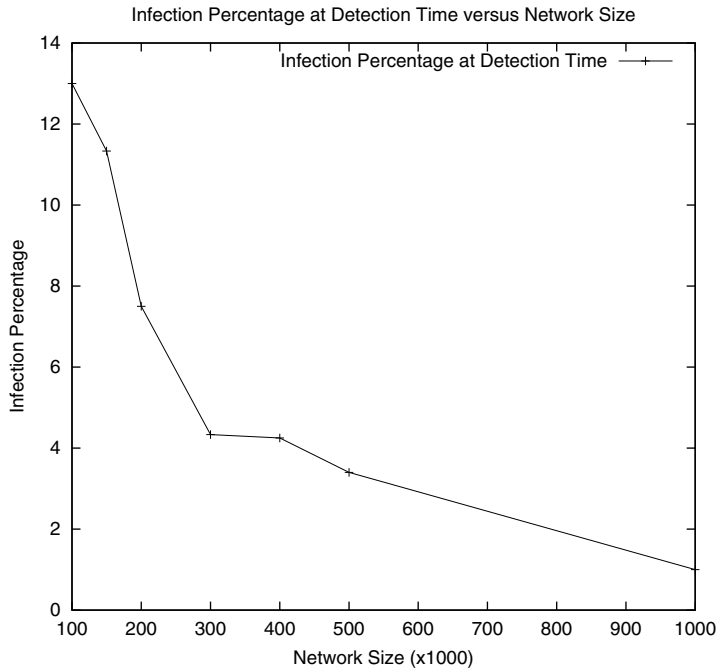


Figure 3.7: Percentage of infected machines at worm-detection time as a function of network size (100,000 to 1,000,000 IP addresses and 100 to 1,000 vulnerable machines); the router coverage is set to a fixed 2 percent.

compared to the size of the address space. Although the number of incoming ICMP-T3s was very large, the chances that one infected system hits the small group of participating routers several times is minimal. Therefore, accurate detection over time requires larger values for Δt. The lower Δt value for Sapphire/Slammer allowed faster simulation runs, but did not affect detection performance. Finally, for simulation convenience, the recovery parameter γ was set to 0.

Figure 3.8 shows the resulting detection performance as a function of router coverage. For 2 class-B instrumented routers (which corresponds to a 0.003% router coverage), Code Red detection occurs at 0.2% infection of the susceptible population, dropping to 0.03% for 16 class-B networks. For 4 class-B networks, Slammer detection occurs at 0.01% infection of the susceptible population, dropping to 0.005% for 16 class-B networks. The drastic increase in detection performance compared to Code Red v2 is due to the vastly increased scanning speed of the Sapphire/Slammer worm, and the smaller number of susceptibles (i.e., more scans were necessary to find one vulnerable system). An important note, however, is that TRAFEN *failed* to detect the Slammer worm with a coverage of 1 or 2 class-B networks, since at these coverages, even the overwhelming scanning activity of Slammer did not cause those routers to generate enough ICMP-T3

messages (due to the ICMP-T3 rate limiting).

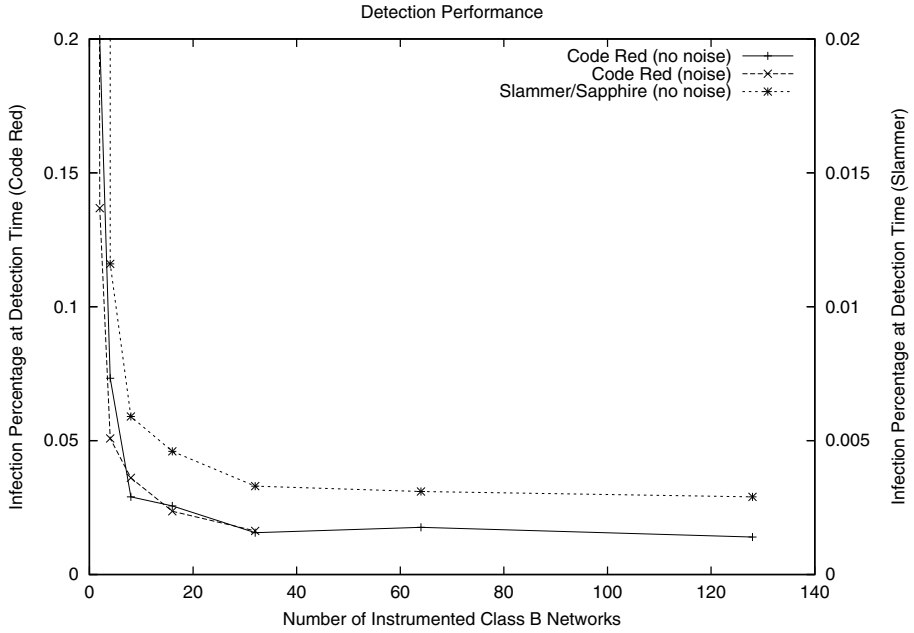


Figure 3.8: Detection performance for the Internet-scale simulated Code Red v2 and Sapphire/Slammer worms.

The simulations for the Code Red v2 worm were run again with a simulated background noise of 1.41 coincidental random probes on the worm's target port per class-B network per second, which corresponds to the background noise observed at the start of the real Code Red v2 worm infection. In other words, participating routers would see, on average, 1.41 unrelated scan packets per second, and thus might generate ICMP-T3s that have no connection with the propagating worm. The results, also shown in Figure 3.8, show that this modest noise level does not affect detection performance. Similar noise results have been obtained for Slammer/Sapphire, although not with the large-scale simulation.

N and Δt. There are many parameters within the DIB:S and TRAFEN systems that affect detection performance. Two of the most important are N, the number of ICMP-T3 messages per generated DIB:S alert, and Δt, the size in seconds of the DIB:S history window. Figure 3.9 shows the detection performance for a small-scale Sapphire/Slammer simulation as a function of N, while Figure 3.10 shows the detection performance as a function of Δt. For both graphs, the network size is 500,000, and the number of vulnerable machines is 500. When N is varied, Δt is held fixed at 300 seconds, and when Δt is varied, N is held fixed at five ICMP-T3 messages per alert. In Figure 3.9, we see that detection performance decreases as N increases, particularly when the router coverage is

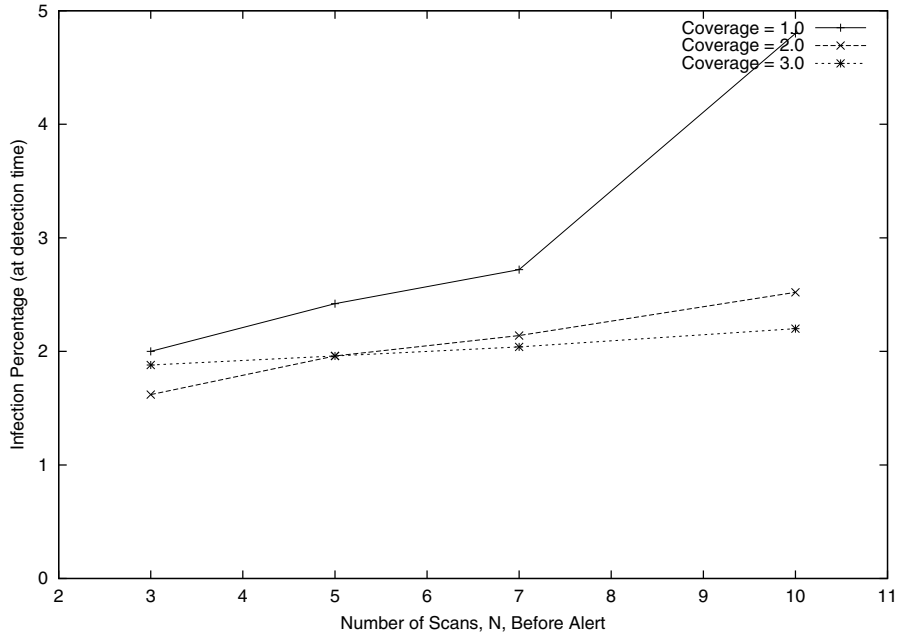


Figure 3.9: Detection performance for different values of N, the number of ICMP-T3 messages required for the generation of a scan alert.

only 1%. At lower coverages and higher values of N, DIB:S might not see enough ICMP-T3 messages to actually generate an alert, and scanning activity will go unreported. In Figure 3.10, we see that detection performance is very poor for the lowest values of Δt, and then after an initial improvement decreases steadily as Δt increases. The very poor performance is due to the fact that when the history window is too small, ICMP-T3 messages will age out before enough messages are received to produce an alert. The steady decrease in performance after the initial improvement is arguably illusory, since when Δt is small, DIB:S will generate multiple scan alerts for the same source address, whereas when Δt is large, DIB:S will generate only one scan alert per source address (during the worm's initial propagation). Although the multiple alerts per source address drive the track probability in TRAFEN above the detection threshold quite quickly, multiple scans from the *same* source address are not, in fact, a reliable indicator of worm activity. They could merely indicate an intense, but manual, scanning effort. In the current system, therefore, Δt must be kept high enough to avoid "duplicate" alerts within too short a time period.

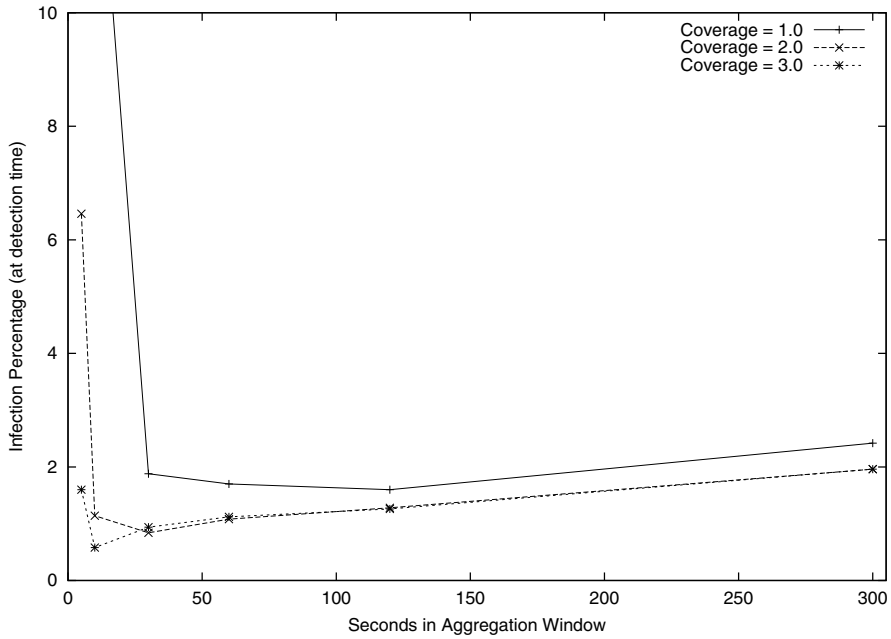


Figure 3.10: Detection performance for different values of Δt, the length, in seconds, of the history window over which ICMP-T3 messages are aggregated.

3.6.3 Discussion

The current TRAFEN model is simple enough that it can lead to false positives. Although our experiments have shown that *random* scanning noise does not affect detection performance, not all scanning noise is random. For example, attackers constantly scan TCP port 80 looking for vulnerable web servers. If many of these scans coincidentally occur within seconds of each other, TRAFEN incorrectly will detect a worm that exploits web servers. Similarly, although lowering N, the number of ICMP-T3 messages needed to detect a scan, gives us faster tracking and detection times, there is a limit to how low N can be. Setting $N = 1$ would result in DIB:S not analyzing anything, but instead passing every unreachable message to TRAFEN. In this case, TRAFEN sees all ICMP-T3 "noise" and has significant problems distinguishing between scanning activity and innocuous ICMP-T3 messages after a normal failed connection attempt. Conversely, when N is chosen too high, DIB:S will generate an accurate view of the world (and will be immune to almost any noise in the ICMP-T3 data), but will generate that accurate view far too late. Choices for Δt are less critical, as long as Δt is high enough that older ICMP-T3 messages do not expire from the history window too soon. TRAFEN presents graphs of current ICMP-T3 activity to system administrators, and the graphs for worms and coincidental scanning activity are immediately and visually distinct, giving direct hints as

how to improve TRAFEN rules: meaning that false positives are less of a problem than they might be in other intrusion-detection applications. We still need to minimize false positives, however, since any false positive involves administrator time. Fortunately, the TRAFEN and DIB:S framework provides significant flexibility for improving the process model. In our case, this means taking into account all of the scanning characteristics of a propagating worm. Scanning activity uniformly distributed in time, no matter how intense, is likely not a worm, but instead simultaneous, but unrelated, attacker scanning efforts. On the other hand, scanning activity that increases linearly or exponentially over time almost certainly is a worm, no matter how much time that increase takes. The goal of the TRAFEN model is to quickly detect an exponential increase in scanning activity (i.e., detect the worm) without incorrectly classifying non-exponential behavior as exponential (i.e., avoid false positives). With the current TRAFEN model and thresholds, detection is best for worms at Code Red v2 speeds or faster. This time dependence was easily removed by submitting multiple worm models to TRAFEN, each focussing on a different infection and spread rate.

The core of the DIB:S system are the routers. Without the routers there would be no data flowing to the DIB:S system. As seen with the large-scale simulation results, a coverage of 4 to 16 class-B networks is enough for accurate detection. Achieving such coverage may be administratively difficult, but is entirely achievable with the cooperation of only a few medium- to large-sized organizations. At this sensor coverage it is possible to detect if there is an exponential increase in the first dozen or so hosts that get infected, meaning that a higher sensor coverage will most likely not lead to a faster detection. It will merely ensure that the detection is more reliable. Alternatively, large portions of the Internet address space are unassigned. If these unassigned address ranges were routed to a system that provided no response to the sender, but merely forwarded appropriate alerts to DIB:S/TRAFEN, we would gain significant data with minimal risk of "noise". Unassigned address ranges never should be contacted in normal Internet communication.

In terms of scalability, if DIB:S is installed at a single, central location, the network bandwidth will limit the number of incoming ICMP-T3 messages. This limit is not as serious as it might appear, however. Even with 64K instrumented routers covering 4 *Class-A* networks, for example, the routers would generate only approximately 200 Mbps of ICMP-T3 messages (at a three per second ICMP-T3 rate limit). In addition, if 200 Mbps is too much network traffic for a single collector site, then DIB:S can be distributed almost to an arbitrary degree. Instrumented routers can send their ICMP-T3 messages to "nearby" collectors, and the analyzers, each of which is in charge of a particular address range, can be distributed throughout the Internet. Even TRAFEN could be distributed by having different copies of TRAFEN handle different sets of destination ports, although this likely is not necessary since the stream of scan alerts coming from DIB:S is significantly smaller than the stream of ICMP-T3 messages flowing into DIB:S.

Additionally, ICMP-T3 messages are not the only data source that can provide indications of worm activity. Although ICMP messages are particularly attractive since they indicate scanning activity that spans multiple independent networks, scan reports and other information from firewalls, intrusion-detection systems and even host-based sensors also can be fed into the DIB:S/TRAFEN system, serving as a useful complement

to the ICMP-T3 data. The ICMP-T3 messages can provide useful additional information themselves, since passive OS fingerprinting[5] would allow DIB:S to infer the type of the operating system that is performing the scan, adding to the hypothesis-generation ability of TRAFEN. Two scans originating from a Linux and Windows machine respectively, for example, most likely do not belong to the same worm.

Finally, regardless of how effective an early warning system is, there is no use in detecting a worm unless something can be done. As discussed earlier, this can be as little as informing system administrators or as much as having a framework in place that will automatically reconfigure firewalls and IDS systems as the epidemic is occurring. Although automated responses may prove more lethal than the worm itself. Even so, early warning is always worthwhile.

3.7 Future of Internet Worms

When we think of the history of computers and the Internet, we have to conclude that history is not a very solid indicator of the future. The way computers and micro-controllers have integrated our lives, through appliances, motor vehicles, communication systems, and personal computers, was completely unforeseen at the time the first computers were constructed with vacuum tubes. It often has been said that truth is stranger than fiction, and this is certainly true in the computing and communication fields. Therefore, it would be pretentious of us to try to paint a picture of the future of computers, the Internet, and their malware. We can make a few inferences, however, based on the worm code of today and proposed techniques for improving worm capabilities.

Computers increasingly are taking on the role of home appliances, integrating such services as game and DVD playing, digital television recording and playback, Internet-based telephony, and traditional personal and home-office computing. In addition, software from a small number of companies is finding its way into more and more products, increasing the likelihood that a software vulnerability will affect a large number of systems. Finally, broadband Internet now is commonplace in many homes, increasing the number of connected systems. With more connected computer systems, often with higher bandwidth, and with more widely deployed software, worm and virus authors will continue to have an ideal environment for their malicious code. There will remain a desperate need for diversity in software and operating systems, decreasing the likelihood of massively homogeneous vulnerabilities.

The increase in connectivity also has prompted a shortage in available IP address space. Although this shortage mostly has been mended with Network Address Translation (NAT), eventually a more structural solution will be needed. Increasing the address space will bring with it the nice property that random scanning for vulnerable IP addresses will become nearly impossible, requiring a significant change in the way authors write their worms. As an example, consider IPv6, which offers 128 bits of address space versus the 32 bits available in IPv4, and Code Red v2, which we analyzed for IPv4 in

[5]Michael Zalewski wrote some of the first passive-fingerprinting code, which is available at http://www.stearns.org/pOf/.

Figure 3.3. We limit ourselves to propagation within a single IPv6 site, which has 2^{64} possible IP addresses. We assume 2^{16} responding machines, of which $1/100^{th}$ are vulnerable. We pick $\gamma = 0$, so that there is no recovery or removal and the worm is free propagate. This makes $r(t)$ a constant, leading to $s(t) + i(t) = M$ being a constant as well, and effectively rewrites the epidemic-model equations [32]:

$$\frac{\mathrm{d}i}{\mathrm{d}t} = \beta si = \beta(M-i)i \tag{3.8}$$

This also is known as the logistic growth equation, and it represents a worst-case epidemic in which there are no recoveries or disconnects, and each infective stays infective forever. Propagation speed will be higher than in a realistic scenario, but the equation allows us to define the absolute limit on propagation speed. Citing Daley and Gani [32] once more for the integral over $(0,t)$, we have

$$i(t) = \frac{i_0 M}{i_0 + (M-i_0)e^{-\beta M t}} \tag{3.9}$$

We can use this formula to find out how fast a worm would spread in the fastest scenario, given ideal connectivity and no countermeasures. To do so, we set

$$i(T_\varepsilon) = \varepsilon M \tag{3.10}$$

where ε is the fraction of susceptibles infected (for example we could define T_{END} by taking $\varepsilon = 0.95$). Replacing the left-hand side of Equation 3.10 with Equation 3.9, and performing straightforward algebraic manipulation – i.e., moving terms to isolate e and then inverting, simplifying, and taking the natural log of both sides – we have

$$T_\varepsilon = \frac{1}{\beta M} \times \ln\left(\frac{\varepsilon(M-i_0)}{i_0(1-\varepsilon)}\right) \tag{3.11}$$

Looking at Equations 3.8 and 3.11, we note two important properties. First, realizing that $O(\beta) \gg O(M)$ and that $O(M) \approx O(i)$, it is clear that the propagation time will be mostly dependent on β. Second, the relationship between propagation time and β is a linear one. If β is doubled, $\frac{\mathrm{d}i}{\mathrm{d}t}$, which is the propagation speed, also doubles. If the speed is doubled, the time it will take for all hosts to be infected will be halved. The linear relationship with β, as well as M, also can be clearly seen from Equation 3.11. Now we can fill in the numbers for Code Red v2 in IPv4 space, assuming the initial number of infected hosts is 10, and we are looking for how long it takes to infect 95% of all susceptible hosts. Remembering that $\beta = 1.23 \times 10^{-9}$ (see the caption of Figure 3.3), we have a time in seconds of

$$T_{0.95} = \frac{1}{1.23 \times 10^{-9} \times 360000} \times \ln\left(\frac{0.95 \times (360000-10)}{10 \times (1-0.95)}\right)$$

which is $30220/3600 = 8.4$ hours, a good approximation of what we can read from Figure 3.3 and thus verifying our equations. Now we fill in the numbers for the Code Red v2 worm propagating within one IPv6 site. First, we calculate r, which is very low,

since the address space, even for just one site, is enormous: $r = 2^{16}/2^{64} = 2^{-48} \approx 10^{-15}$. Next, we obtain τ by filling in Equation 3.2: $\tau = 10^{-15} \times 1 + (1 - 10^{-15}r) \times 21 \approx 21$. Finally, we obtain β by filling in equation 3.3: $\beta = \frac{1}{2^{64}} \times \frac{100}{21} \approx 2.5814 \times 10^{-19}$. With $M = 2^{16}/100 \approx 655$, the time to reach 95% propagation is:

$$T_{0.95} = \frac{1}{2.5814 \times 10^{-19} \times 655} \times \ln\left(\frac{0.95 \times (655 - 10)}{10 \times (1 - 0.95)}\right)$$

which gives 4.2057×10^{16}, or over 1.3 billion years, and confirms the intuition that an enlarged address space will pose a significant challenge to randomly propagating worms.

This undoubtedly will lead to new and improved target selection techniques, most of which were already discussed in the worm propagation Section 3.3.2. We will mention two of them again, however, and suggest probable detection strategies. To acquire IP addresses of hosts running a vulnerable service the worm could sniff the network wire for traffic from that service. Mail and DNS servers will be most vulnerable to this approach, since they constantly communicate between peers. Imagine a worm that moved from web browser to web server, and from web server to web browser. Similar worms could be designed for any peer-to-peer or client/server service (as long as the clients regularly communicate with different services). One possible way of detecting such a worm is by inserting bogus communication into the network. By spoofing non-existent IP addresses and so making fake queries to all the services in the network, sniffing worms can be provoked to connect to these non-existent machines. The challenge would be to make the fake communication look as real as possible, ensuring that the worm could not distinguish between real and false events. Worms attempting to connect to the non-existent addresses would provoke ICMP-T3 messages, which could be fed into the DIB:S/TRAFEN system.

The DIB:S/TRAFEN system also can be used in the case of DNS exploration. As noted before, worms can gain hostnames by probing DNS servers and potentially trying whole ranges of possibly related hostnames (recall the example with *sparc01, sparc02, ... sparc99*). DIB:S could be configured to receive notification of all failed DNS queries, as a blind carbon copy from name servers. The analogy is simple: one IP address attempted to contact many *hostnames* on many different networks (and failed). This would be a clear bloom, and TRAFEN soon would detect the worm when multiple hosts show the same behavior. The analogy ends, however, when we try to distinguish between multiple worms that are using DNS queries to obtain IP addresses. There will be no significant differences. To tackle this problem, a more radical approach is needed. DNS servers could be configured to respond with unassigned IP addresses when presented with repeated DNS requests for unknown hostnames. This way worms would attempt to connect to the targeted service port on unassigned IP addresses, leading to generation of ICMP-T3 messages suited for DIB:S. The drawback is a long timeout when users mistype hostnames, since the DNS server might give an unassigned IP address instead of an error message.

Finally, worms will begin to use some of the same polymorphism techniques as the most advanced viruses, such as encrypting and permuting basic code blocks on each propagation, making signature-based detection more difficult. Thus, inferring the exis-

tence of worms through their secondary network traffic (such as ICMP-T3 messages), rather than using signatures, always will be an important detection strategy, even for *previously seen* worms.

Companies that today provide automated virus-signature update services will continue to do so, will add worm signatures, and possibly will provide services for content-based filtering in transit. To elude detection during propagation, however, worms can be created that use some of the same polymorphism techniques as the most advanced viruses. The easiest way of creating polymorphic code is by encryption. A small section of code can decrypt the entire program using a key that is propagated along with the code. When such a worm found a vulnerable target, it would create a key, then encrypt itself, and propagate both the key and the encrypted version of its code. On the newly infected machine, the key would be used to decrypt the worm and start the procedure of looking for new vulnerable hosts. When the initial decryption code is relatively small, it might be very difficult to create a proper signature for such a worm. An alternative for a polymorphic worm would be to permute the basic code blocks before each propagation. These blocks are linear sets of instructions between branches or jumps, and can be moved to arbitrary locations when all the corresponding branch instructions are updated properly. Signatures can no longer be singular in such a case, and would need to include several sufficiently large basic blocks and a query that checks if all those blocks are present, regardless of their order. In short, inferring the existence of worms through their secondary network traffic (such as ICMP-T3 messages), rather than using signatures, always will be an important detection strategy.

3.8 Conclusion

Most current worms identify vulnerable machines through random probing of the address space, as the Internet becomes more and more densely populated with machines, such worms will be able to spread faster and faster. Fortunately, it is possible to quickly detect such worms by looking for unusual patterns in different kinds of network traffic. In this chapter, we explored the use of ICMP-T3 messages for worm detection. When a connection request is made to an IP address that is not populated by an actual system, routers along the path may return ICMP Destination Unreachable messages (ICMP-T3). The system we developed, DIB:S/TRAFEN, collects ICMP-T3 messages forwarded from participating routers, and looks for the distinct, bloom-like connection pattern that worm-infected hosts exhibit while they are randomly scanning for targets. Using both small-scale and large-scale simulated worms, we demonstrated that our system is capable of detecting propagating worms early in their lifetime. In particular, the large-scale simulation indicates that a router coverage of 16 class-B networks is enough to detect worms that spread at Code Red v2 and Sapphire/Slammer rates before 0.03% of the vulnerable machines are infected. These results, particularly since they involve a router coverage that would be achievable in the real Internet, are extremely promising. When DIB:S/TRAFEN is fully deployed on the real Internet, it will be able to detect active worms early enough to take meaningful defensive action.

At the same time, however, there is a lack of actual active-response capability for the current Internet. Although there has been some initial work in this area, significant additional research is needed to develop field-deployable solutions. Moreover, we can expect worm authors to continue to improve their target-selection techniques, and we can expect to see many worms that use alternatives to random scanning. Some of these future worms could be detected by looking for other patterns of unusual network activity, possibly through DIB:S/TRAFEN, but some will require entirely new approaches. For example, imagine a contagion worm that starts on a web client, spreads to any vulnerable web servers that *visit* that web client, spreads to any vulnerable clients that visit those infected web servers, and so on. It is not clear that such a worm would generate unusual patterns of network activity, Instead such worms might need to be "tricked" into attempting to infect a dummy server or client whose entire purpose is watching for infection attempts. Many other techniques are both possible and necessary. Additionally, as many other authors have concluded, it is important to note that diversity in operating systems and server software, as well as appropriate maintenance and patching procedures, mitigates the total damage that any individual worm can do.

Finally, the performance in terms of processing power and memory footprint is very good, thanks to the tight integration of the DIB:S software. However, if the ICMP-T3 sensor data was directly sent to a general-purpose PQS system, running a DIB:S-like PQML model, then processing power would have been constrained by the PQS system, probably to several hundres of ICMP-T3 observations per second per PQS-instance. In a large-scale deployment this would require running multiple PQS-instances, and possibly using compiled models, instead of PQML models (which is essentially what DIB:S does by using multiple analyzers). Therefore, difference between the DIB:S system and an actual general-purpose PQS system is that in a PQS the expert knowledge should only be integrated into the models, not the datastructures, as it is done with DIB:S today.

Chapter 4

The PQS-Net System

4.1 Introduction

The PQS-Net system is a valuable proof-of-concept that the PQS framework is ideally suited to do high-level network security monitoring on large, enterprise-class networks. Enterprise-class networks are split up into many different subnets, often geographically diverse in nature. These subnets are then connected through WAN connections, allowing the network to spread worldwide. Enterprise networks may contain as many as 2 million connected hosts, many of which will be mission critical servers. Any form of central security monitoring will be very difficult. Standard intrusion detection sensors such as *Snort* [107] have a tendency to generate large quantities of alerts, most of which are false positives, therefore often obscuring the important information. Most notably, they only collect data for the subnet that they are connected to. Needless to say, security administrators need ways to collect this, and other data from the many networks at different sites, and be able to sift through the data in a meaningful manner.

In addition to Snort IDS logs, security administrators will often also consult services such as *Dshield.org* [35] to get an idea of which attacks are most popular at the current time. Combined with *server logs*, *system logs*, and *TripWire logs* [121] collected from the many sites, security administrators will have a flood of data to review. It is not difficult to see how this data can grow up to several gigabytes per day that need to be analyzed, most of which will be false positives.

The PQS-Net system collects data from many different sensors and implements PQS models based on methodical procedures commonly followed by security administrators. These models evaluate the data coming from the many sensors and the conclusions are published though a web interface. The system eliminates most of the false positives, focussing on detecting the processes that are indicative of malicious hacker behavior, insider threat behavior, and autonomic attacks such as worms and viruses. To improve the power of the PQS-Net system, models were also written that specifically search for network failures, such as faulty routers, or misconfigured servers. This way the system is able to disambiguate between malicious hacker attacks, network failures, and benign behavior.

To test the PQS models and verify the results a test network was constructed that is complete with several subnets, network servers, and actual user hosts. Since the setup of this network is important for the understanding of the PQS-Net system, the next section will focus on this basic infrastructure. The sections afterward will focus on the sensors that were deployed, and the models that were written. Finally, this chapter will close with a discussion of our experiences.

4.2 Infrastructure

Figure 4.1 shows a toplevel schematic view of the network that was constructed for the PQS-Net project. Although the network is small compared to an actual enterprise class network, it features a wide range of systems and servers that are typical of larger organisations. Looking at the models it will become evident that the size of the test network

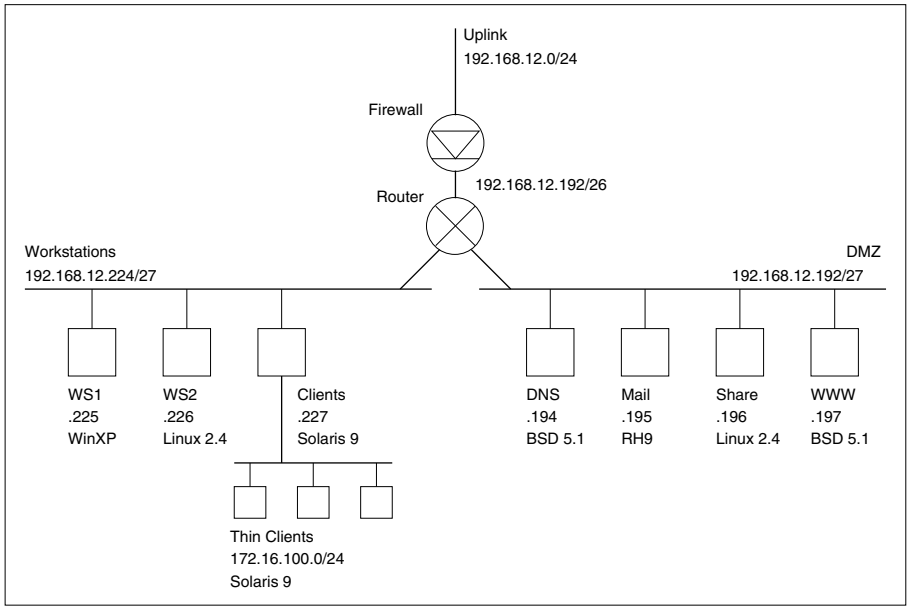


Figure 4.1: Schematic overview of the most important systems in the PQS-Net network.

is irrelevant because of the scalability of the PQS framework. As more hosts are added the number of observable events increase, however, were the load on any one single PQS instance to become too high, it is always possible to spread the load over multiple instances. Several models are ultimately suited to only monitor the local subnet, for example, therefore creating good opportunities for subdividing the PQS instances.

Behind the firewall there are 64 addresses available (.192/26), which are split up between a *Workstations* network and a *Demilitarized Zone* server network, both (/27s). The uplink connects directly to the Internet, and all addresses are globally routable[1]. Both subnets contain more hosts than are displayed in the image, however; they are immaterial to the dicussion below and have therefore been omitted. The workstations network features several Windows XP clients, Linux 2.4 and 2.6 systems, several Solaris 9 workstations, and a Solaris 9 server to which multiple thin-clients are connected. All these systems are used daily by students, and so the traffic on this network is typical for a normally operating organization where people browse the web, print documents, and download files during business hours.

The DMZ features several servers implemented on several different operating systems. The *DNS* server resolves names for all the hosts connected to the network, *Mail* handles inbound and outbound email traffic for the students, and *WWW* serves up the results produced by PQS-Net system, as well as personal webpages. Finally, the *Share*

[1]The first three octets of these addresses have been obscured by replacing them with unroutable addresses.

server holds home-directories, as well as general fileshare areas that can be used to store and backup data.

The firewall is configured to allow nearly all inbound traffic in, and only protects the core PQS infrastructure, which is not considered part of the test network, and therefore not shown in the image. Although this would be uncommon for a normal organization, it does allow a lot of scans and attacks through that naturally come from the Internet [88, 73]. These scans form a nice baseline to test the PQS-Net system on actual malicious traffic. In addition to the expected malicious traffic coming from the Internet the PQS-Net network is also attacked from dedicated *Attack Systems* that are located both inside the network, as well as outside on the Internet. These attacks include network scans, vulnerability scans, and directed attacks, all intended to test the PQS-Net detection capabilities.

4.3 Sensors

In order to apply a *Process Query System* to any area the programmer must do two things: (1) connect the sensors, and (2) implement the process models. This section describes the first step; what sensors are installed in the test network and connected to the PQS system. Generally, the sensors can be split up into three categories: *Global* (Internet wide data), *Network* (network specific data), and *Hostbased* (data pertaining to one specific host). In the subsections below all connected sensors are outlined, together with the data that they provide to the PQS system. Figure 4.2 shows where the sensors are located in the test network. The placement of the sensors is roughly similar to the common data collection points that security administrators use.

4.3.1 Global

Both the DIB:S (Dartmouth ICMP Bcc: system) sensor, as well as the BGP (Border Gateway Protocol) sensor are not directly connected to the test network as shown in Figure 4.2. Because of their global nature; their data collection and conclusion generation occurs outside of the test network. The PQS system merely connects outbound to both of these sensors to get their reports. Global sensors are important because they provide a solid comparison with "trends"; i.e. "can the locally collected data be explained by events seen throughout the Internet, or is the local network specifically under attack?" Essentially, the global sensors fulfill the role of network security sites such as *www.dshield.org*, which security administrators may refer to regularly. The PQS models then have the ability to compare locally gathered data with Internet trends.

- *DIB:S* For the PQS-Net system DIB:S (see Chapter 3) output conclusions are taken as sensor input into TRAFEN. As a global sensor, DIB:S reports on scanning patterns throughout the Internet in real time. The most general form of information coming from this sensor is: "port *P* gets scanned a lot everywhere". DIB:S, however, does provide more specific information regarding scanning and frequently

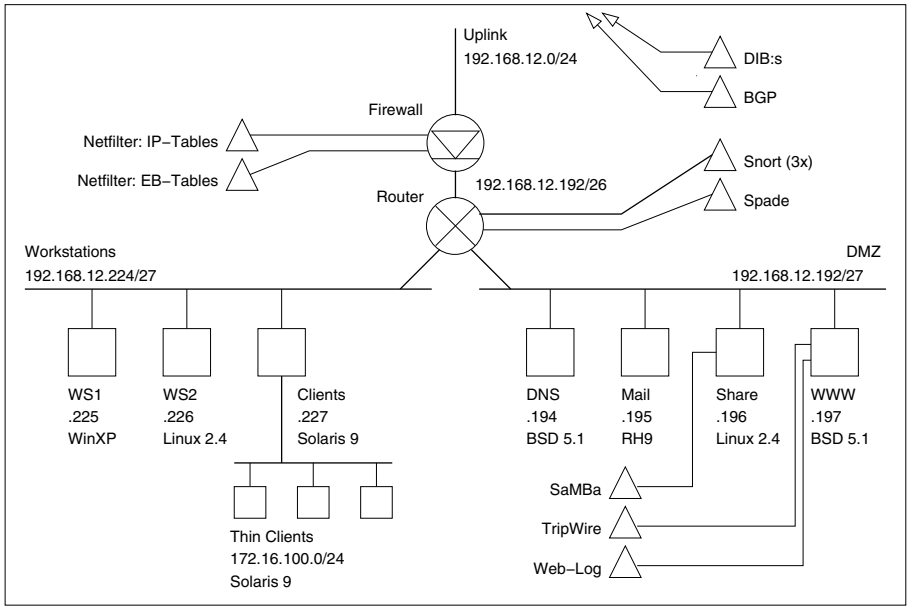


Figure 4.2: Overview of the location of sensors in the test network.

scanned hosts, therefore allowing conclusions such as: "our network gets significantly more probes on port P' than the rest of the Internet", which could be a significant conclusion, because the network may be specifically targetted. The specific format of the observations generated by the DIB:S sensor is (note that the data formats here are in sensor-output format, the PQML model will handle all *char*, *short*, and *long* as *int*, and all *struct timeval* as two *int*):

```
struct timeval ts;
long           src_ip;
long           dst_ip;
char           proto;
short          sport;
short          dport;
```

which means: In the last Δt seconds, on the same port p (*sport*) and using the same protocol P (*proto*), one host (*src_ip*) has contacted N different IP addresses (Case 1), or one host (*dst_ip*) has been contacted by N different IP addresses (Case 2). Where Δt and N are configured in the DIB:S system, for the PQS-Net system: $\Delta t = 600$ seconds and $N = 4$ hosts.

The DIB:S sensor works in collaboration with dozens of routers distributed across the Internet. Whenever an unsolicited request comes in, the router returns an

ICMP-T3 (Destination Unreachable) message to the originator of that packet. It also, however, forwards a second copy of this ICMP-T3 to the DIB:S system, which then does book-keeping and statistical analysis on these messages to determine which hosts are scanning, and which are being scanned. Note: the DIB:S architecture is described in-depth in Chapter 3.

- *BGP* The Border Gateway Protocol is used between top-level routers on the Internet. At the highest level the Internet is divided into *Autonomous Systems*. An autonomous system (AS) is an enclave of associated networks; for example, a college campus would be considered one autonomous system, consisting of many class-C networks. Technically an AS is a collection of networks with the same prefix, defined by their CIDR notation. CIDR, or Class-less Inter-Domain Routing blocks are identified by their network base address, a slash, and the number of fixed bits in the network address.

  Consider the following CIDR block: 129.170.0.0/16, where 129.170.0.0 is the *base network address*, and 16 is the number of fixed bits in the address. In this case 16 bits are the first two octets (129 and 170), meaning that any address that starts with 129.170 is part of this CIDR block. So 129.170.249.24 is part of the block, although 129.171.249.24 is not, because the second octet is not 170. When the fixed number of bits is a multiple of 8 a CIDR notation is quite intuitive because each set of 8 bits in a network address is another octet. A class-C network, for example, can be designated as a *slash-24* CIDR block: 129.170.249.0/24 would mean all addresses that have 129.170.249 as the first three octets. The last octet is variable, meaning that a class-C network contains 256 addresses ($2^8 = 256$). Specifically, because there are 32 bits in every IP address in IPv4 Internet addressing the number of IP addresses N in a CIDR block can be calculated as $N = 2^{32-s}$, where s is the number of fixed bits in the CIDR-block. Another example was given by Figure 4.1 where the 192.168.12.192/26 network was split up into two slash-27 networks: 192.168.12.192/27 and 192.168.12.224/27. The slash-26 network ($2^{32-26} = 64$ addresses) was split up into two slash-27 networks (each $2^{32-27} = 32$ addresses).

  The BGP-4 protocol [101] determines how packets are to be routed *through* autonomous systems by defining how these systems are interconnected. The BGP protocol allows a top-level router to announce to all other top-level routers how its AS is connected to other autonomous systems. Specifically, such an announcement would include the CIDR blocks that are part of its AS, and a list of other autonomous systems to which it is connected. This way each top-level border router will collect lists of all autonomous systems that it can reach, each with a list of other border routers to traverse. Such a list would look somewhat like this:

```
129.170.0.0/16   245 21 1103 554
24.128.16.0/20   245 915 582 45
64.151.135.0/24   12 873 16
```

where the first column is the CIDR block and the following list contains the AS numbers to traverse for reaching the top-level border router responsible for this CIDR block. Therefore, when a top-level border router needs to route a packet to a specific destination it will look in this table to determine what AS the destination is a member of. Knowing the destination AS it knows what route the packet needs to take to get to the destination IP (essentially: which AS it will send the packet to, next. After that it becomes the responsibility of the next border router, which will follow the same steps).

Knowing that physical connections are changed constantly (due to new connections being made, wires breaking, satellites failing, etc.) these tables need to be updated constantly in all the border routers. These announcements "update" the reachability of CIDR-blocks. When the Internet is under heavy load (for example due to worm propagation) it is observed that the number of route updates increases drastically. This is usually due to top-level border routers failing under heavy load, therefore making entire CIDR blocks unreachable. The BGP sensor collects these update messages from hundreds of top-level border routers and calculates how much of allocated address space on the Internet is reachable. This number is referred to as the "Global Reachability Index" and is normally between 0.98 and 0.99, indicating that between 98% and 99% of the allocated addresses on the Internet are reachable. The second measure that is generated by the BGP sensor is the portion of the CIDR blocks that had their routes updated in the last 24 hours. This is referred to as the "Global Instability Index". This number is normally expected to be between 0.08 and 0.12, indicating that about 8% to 12% of the CIDR blocks on the Internet have their routes updated per day. The format of the messages is therefore:

```
double  GRI;
double  GII;
```

By itself this information may not lead to definitive conclusions, however, combined with other forms of information it can strengthen or weaken existing hypotheses. For instance, when parts of the Internet are unreachable it could indicate both local router failure, as well as general higher-level routing problems. The PQS system will generate both hypotheses. The local router failure hypothesis may be weakened in favor of the higher-level routing problems hypothesis by a BGP observation with a GII of 90% and a GRI of 20%.

4.3.2 Network

Network sensors are the most common class of sensors, often having one or more instances of each per local network. In particular, it is not uncommon to have a copy of the Snort IDS running on every segment of the network, thus scanning all traffic for signatures of malicious activity. This brings with it two artifacts that a user has to account

for when building models for a PQS. The first is the fact that observations from the same type (e.g. Snort alerts) may have come from different sensors. For network sensors this is often not an issue since most (if not all) observations pertain directly to a given IP address, therefore making the detection point irrelevant. The second artifact may lead to more confusion: two sensors both generate an observation based on the same cause. Models must account for the fact that both observations may have been generated due to the same event. Consider, for example, a packet enters the network carrying a malicious payload. En route to its destination it traverses several local network segments, where each network segment hosts a Snort IDS sensor. The PQS system will then see observations from several Snort IDS sensors, all containing the same alert, generated just a few microseconds apart. Some models may have to be aware this situation, while for others it may not matter at all.

- *Snort* This sensor is based on the freely available network intrusion detection system Snort. Commercial versions will be just as effective, if not more so. The general concept of network intrusion detection systems is to do elaborate signature matching on all packets flowing past it. This often means that each packet needs to be compared to thousands of known malicious signatures in the database. Whenever a packet (or a set of packets) matches, an entry is written into the log file. Most PQS-Net sensors are based on a template that allows for easy parsing of dynamically growing log files. Using this template the Snort sensor retrieves the following fields:

```
struct timeval  timestamp;
int             ruleNum;
int             srcip;
int             dstip;
char            proto;
short           srcport;
short           dstport;
int             prio;
```

  where *ruleNum* refers to the rule that matched the packet coming from *srcip:srcport* going to *dstip:dstport* using protocol *proto*. The *prio* field returns the priority (or severity) associated with the particular rule that was matched.

  The most notable messages obtained from an IDS are specific exploits, network scan reports, and Trojan/backdoor traffic. Often the systems that are the target of the exploit are not vulnerable to that exploit. Because the IDS has no way of knowing if an attack was succesful, the false positive rate is usually very high. Models in the PQS-Net system match these IDS alerts with other indicators of possible host infection, such as hostbased sensors like TripWire and weblog files, or other Snort alerts. This way most (often all) of the IDS false positives can be eliminated.

- *Spade* A significant drawback of intrustion detection systems is their inability to detect unknown attacks. Although this is a problem that is hard to solve, it can, however, be mitigated to some degree. The Spade Anomaly Detection sensor is a plugin to Snort, although the concept is general and does not depend on an IDS [113]. All packets that pass through a network are categorized by some identifyer (e.g. destination IP and port). Next, a count is kept for each category, identifying how many packets of that type were observed in a given timeframe (usually a week or so). These counts then give a probability that a packet of a certain type will be seen on the network. The anomaly score is the inverse of that probability.

  Consider, for example, the PQS-Net DMZ with a DNS server, an email server, a fileshare, and a web server. Because of their tasks the DNS server will get a lot of UDP 53 requests, and the web server TCP 80. Therefore the probabilities for these kinds of packets will be high, and the anomaly score low. However, when a TCP 80 packet is sent to the DNS server there will probably not even be a table for that type in the anomaly detection system; the probability will be near zero and the anomaly score will be very high. This in itself may not be a reason for concern, but once again it is about the bigger picture: what are other sensors reporting. If, for example, shortly after this unusual packet was seen, the DNS server starts to communicate on highly unusual ports, also triggering high anomaly scores, the server may have been compromised.

  The specific anomaly score calculator in Spade categorizes packets by their destination IP and port. For any packet x the anomaly score is calculated as $A(x) = -\log_2(P(x))$. The observation received by the PQS is:

```
struct timeval  timestamp;
int             srcip;
int             dstip;
char            proto;
short           srcport;
short           dstport;
float           anomaly;
```

  where *anomaly* is the anomaly score for the packet. Optionally, since the sensor will produce an anomaly score for *every* packet on the network, a threshold can be set to only report the $x\%$ most anomalous packets.

- *Netfilter IPtables* Both the IPtables as well as the EBtables sensors are based on the Linux Netfilter firewall [84]. The general concept behind this class of network sensors is to report on known incorrect network behavior. For example, DMZ machines are *never* supposed to contact workstations. If this happens it could be a evidence of a compromised server. Also if an email server starts doing anything more than sending email or resolving names with the local DNS server, it may be

compromised. Although another explanation would be that the server is improperly configured; this is something for the PQS models to figure out.

In the PQS-Net network the IPtables sensor is configured to log any traffic from the DMZ outbound to the Internet which is not directly associated with the task of a particular server. Meaning that the only traffic which is not logged is email connections from the mail server and name resolutions from the DNS server (the web server should only be contacted inbound by clients, and the fileshare is accessible only from inside.) Additionally, communications with known suspicious port numbers is also logged, for example port 31337 (Backorifice [4]) and port 27374 (Subseven [119]), both associated with well-known Windows trojans. The observation received from the IPtables sensor is (note that the *char** type maps to *string* in PQML):

```
struct timeval  timestamp;
char            *hostname;
char            *comment;
int             src_ip;
int             dst_ip;
int             len;
short           proto;
short           src_port;
short           dst_port;
```

where *hostname* is the hostname of the firewall that generated this observation, *comment* is the comment field associated with this rule in the firewall, and *len* is the length of the packet reported by this observation.

- *Netfilter EBtables* This sensor produces exactly the same observations as the IPtables sensor, however, the underlying Netfilter implementation is built on a bridge, instead of on a router. This means that this firewall sensor is able to pick up non-routable layer 2 packets in addition to the usual layer 3 traffic. Specifically, this sensor is configured to report *only* on ARP traffic. Often a flood of ARP packets can be a strong indicator of network misconfigurations. Consider, for example, a workstation is persistently looking for the MAC address associated with unused IP address 192.168.12.233. This is usually an indicator of some software misconfiguration of the host. If the host is sending out ARP requests for all possible addresses on the subnet it is likely an indicator of scan behavior, meaning that the host may have been compromised.

4.3.3 Host

Hostbased sensors provide specific information pertaining to one host. Their focus is on information that cannot be gathered from network traffic alone.

- *Tripwire* When a computer system is compromised the attacker will often want to be able to re-enter the host at a later time, either to steal information or launch another attack. Therefore the attacker will need to leave a *backdoor* in the form of a software program that listens for an incoming connection from the attacker, or possibly connects back to the attacker at a predetermined time. Because such a backdoor program is easily detectable by administrators using common system tools, the attacker will try to hide his or her presence to avoid being noticed. The most straightforward method is to disable these system tools, or to modify them such that the backdoor software is undetectable. Then, when the system administrator uses these tools the backdoor program will no longer show up.

  Tripwire [121] is one of several "host filesystem integrity checkers", a class of software programs that keeps a database of MD5 checksums of a large number of important system tools and libraries. This database is built right after installing Tripwire (assuming the system is not compromised yet), and then used every 12 hours or so to check all the monitored files. This is done by regenerating MD5 checksums of these files and comparing the result to the checksum stored in the database. Since a single bit-change in a file will change the MD5 checksum completely, even the slightest modification of system tools will show up right away. For example, on a Unix system the files that Tripwire monitors include: /etc/passwd, /bin/ls, /bin/ps, /bin/login, etc. The output of this sensor is:

  ```
  int            type;
  struct timeval ts;
  char           *hostname;
  char           *filename;
  ```

  where *type* is either: 1. added, 2. removed, 3. modified, *hostname* the host that this observation came from, and *filename* is the file that was modified.

- *Samba* This sensor was chosen for its platform independence. The concept will work for any type of filesharing method. Samba is the name given to the SMB protocol (Server Message Block), which is the default and primary way in which Windows computers share resources over the network. Clients are available for all major operating systems. The idea behind this sensor is to monitor which user accesses which files at what time, and from where. This leads to very verbose information since users are accessing files on a fileshare all the time. To improve the performance of this sensor it is recommended that only the most important files are monitored. When one of the monitored files is accessed (read, write, or modify) this is reported by the sensor in the following format:

  ```
  struct timeval  ts;
  int             src_ip;
  char            *filename;
  char            *smbname;
  char            *user;
  ```

where *src_ip* is the computer that was used to access *filename* on fileshare *smb-name* by user *user*. It is up to the PQS models to verify if anything is out of the ordinary.

To make this sensor even more powerful it is possible to distribute some files across the directories on the share that do not contain any useful information. These files function as bait for possible attackers or malicious insiders and they are intentionally "booby-trapped". As soon as any of these special files are opened it is immediately considered an access violation. These special files are referred to as "honey tokens" and are intended to attract the attention of an attacker.

Most of the above sensors are either generally present in a network, or can be very easily deployed. The goal of the PQS-Net project was, therefore, to design a system that uses a conventional, every-day sensing infrastructure. Since a PQS takes care of all data processing and hypothesis generation, only the sensors need to be hooked up and the models need to be submitted. The models, therefore, contain the expert domain knowledge.

4.4 Models

The second part of applying a Process Query System to a specific task consists of defining the models that are to be submitted. In the construction of the PQS-Net system many models were written; this section outlines several of these models, separated into attacks, network failures, and higher level models. Recall that models are asked by the PQS to "score" sequences of observed events. If the sequence of events is evidence of the process that the model is describing, then the score should be high, otherwise it should be low. Some of the earlier models were extremely complex and needed a lot of tuning. We quickly learned, however, that the power of PQS encourages simpler models; either two or three process states, or just several decision rules.

Since a PQS automatically creates all possible hypotheses at any given time, it is important to structure the models such that very unlikely or impossible combinations of observations are scored low enough such that they will be pruned away quickly. The general rule of thumb for creating models in a large system is to *encourage* promising combinations of observations (with a "higher" score), and to *discourage* poor combinations of observations (with a "lower" score).

Most of the sensors discussed above (Snort, Spade, Samba, ...) have a tendency to produce a large flood of observations. It is the task of a good model to sift through this and find relevant combinations of events that are evidence of the modeled process. Additionally, some observations have more "weight" than others. For instance, consider a stream of Snort scan alerts towards the web server versus a single firewall observation of unexpected outbound traffic from the web server. The stream of Snort scan alerts is not necessarily alarming, considering that any Internet-connected DMZ servers will experience regular scanning, however, unexpected outbound traffic from the web server is almost always related to a compromise. Therefore, based on the firewall observation,

the Snort scan alerts become more important and can be used as evidence to support that the web server may have been compromised. (Note: the DIB:S based worm models are described in-depth in Chapter 3.)
Terminology:

$S_{\mathcal{M}}(T_n)$	New score of track T at time n given by model $\mathcal{M}$.
$S_{\mathcal{M}}(T_{n-1})$	Old score previously assigned to this track by model $\mathcal{M}$.

4.4.1 Attacks

Often the observed events can be explained in various different ways. For instance, DIB:S observations may indicate that the network is being scanned, or that a server has failed. Similarly, Tripwire observations may indicate that a system was compromised and important files were changed, however, it can also indicate that a system software update was performed recently. Because of the multiple hypothesis capability of the PQS system it is possible to keep both hypotheses around in the above cases (given that the PQS is configured to keep several hypotheses around). As more observations come in the models may favor one hypothesis over the other. In this way we load multiple models into the PQS and allow them to contend for observations, while forming a conclusion. This section describes several succesful attack detection models.

Noisy Worm Propagation

sensors: DIB:S
Snort
BGP

This model looks for worms that show aggressive scanning patterns. The most typical and early evidence for worms comes from the DIB:S system (see Chapter 3). During the initial stages of worm propagation an exponential growth in infected systems is expected. Since DIB:S reports on hosts that display massive parallel scan behavior it is expected that the number of DIB:S alerts for the destination port of the worm will also increase exponentially. A track consisting entirely of the same destination port DIB:S alerts gets scored as follows:

$$S_{\mathcal{M}}(T_n) = \begin{cases} \frac{1}{2} \times (1 + S_{\mathcal{M}}(T_{n-1})) & : \quad \Delta t < 10 \\ \frac{1}{2} \times (\frac{\Delta t - 10}{890} + S_{\mathcal{M}}(T_{n-1})) & : \quad 10 \leq \Delta t \leq 900 \\ \frac{1}{2} \times S_{\mathcal{M}}(T_{n-1}) & : \quad \Delta t > 900 \\ 0 & : \quad \text{when destination ports mismatch} \end{cases}$$

where Δt stands for the time between two consecutive DIB:S alerts in the track. The above function alone is a very powerful worm detection model when submitted to a PQS. The score is constructed for 50% from the previous score that this model assigned, and for 50% based on the time between the newly added observation and the previous observation. This weighting effectively creates a filter that smooths out the scores over

time, and is the same as the model that is used in the original DIB:S/TRAFEN system described in Chapter 3. This model is specifically tuned to to worms that propagate at a speed typical of reaching maximum infection rates within 30 minutes to several hours. To improve the model and increase the amount of information returned, the following functions may be added:

$$S_{\mathcal{M}}(T_n) = \begin{cases} \frac{1}{6} \times (1 + 5 \times S_{\mathcal{M}}(T_{n-1})) & : \text{Snort port and protocol match} \\ \frac{1}{6} \times (1 + 5 \times S_{\mathcal{M}}(T_{n-1})) & : \text{BGP GII} > 12.0 \\ 0 & : \text{otherwise} \end{cases}$$

However, all tracks must be started, and primarily be constructed from DIB:S observations. The rule for Snort observations encourages tracks when there are matching Snort rules for the vulnerability that the worm is exploiting. The BGP observation may be expected later into the propagation of the worm where the network load on the Internet is causing routing instabilities (a GGI$>$ 12.0). In this calculation the previous score that this model assigned to the track is weighted in 5 times. Finally, the score of any new, single-observation DIB:S tracks is set to 0.1 for this model.

Email Virus Propagation

sensors: IPtables
DIB:S
Snort
Spade

This model is only concerned with local systems infected with an email virus. Two things are checked, first: are there any systems other than the email server sending SMTP (TCP/25) traffic outbound, and second: how fast is this occurring? Both models must be considered before a conclusion can be made. The IPtables sensor is specifically instrumented to report on any *outbound* SMTP traffic. The score is determined as follows for all observations:

```
if (local(src_ip) AND proto == 6 AND dst_port == 25)
then
    CalculateScore()
else
    Return(0)
```

When a zero is returned it will discourage this track from growing any further in the PQS system. However, this does not mean that there are no others tracks forming that *do* provide evidence of an email virus. When the above truth statement is true, the score is calculated as follows:

$$S_{\mathcal{M}}(T_n) = \begin{cases} \frac{1}{2} \times (1 + S_{\mathcal{M}}(T_{n-1})) & : \Delta t < \mathcal{T} \\ \frac{1}{2} \times (0.1 + S_{\mathcal{M}}(T_{n-1})) & : \Delta t \geq \mathcal{T} \end{cases}$$

where Δt is the time between two email messages, and $\mathcal{T}$ is the threshold time between growing or shrinking the score. This model is simple, yet very effective. The only parameter is the threshold, which can be adjusted to match normal levels of email traffic on the network. If email messages are being sent out *generally too fast* then the likelihood that an email virus has infected the network quickly grows to 1. If the rate at which messages are sent is *generally lower* then this likelihood shrinks to 0.1, at which point the track runs a high risk of getting pruned by the PQS. New tracks may start with all four observation types, and get an initial score of 0.3 when the truth equation matches.

This model is a typical example of a very robust process. The specifics on the threshold, *too fast*, *too slow*, and *high score* or *low score* do not matter very much. Although the model does perform a little bit better when it is properly tuned, it also performs very well when these values are chosen by best guess. The reason for this is the inherent flexible nature of a PQS; decent models tend to perform very well when part of the previously assigned score is embedded in the newly assigned score.

Remote Administration Tool Deployment

sensors: DIB:S
Snort

This model searches for evidence of Remote Administration Tool (RAT) deployment. Remote administration tools are often characterized by Trojan backdoors, such as *Subseven* or *Backorifice*. For most of these tools there exist Snort IDS rules that recognize signatures of such RAT traffic. Since it is very likely that an attacker will attempt to install backdoors on many hosts, hoping to infect at least a few, the attacker's behavior will often also show up in DIB:S observations. This model sometimes seems "trigger happy" because many forms of malicious behavior tend to initially show up as RAT deploying hosts. Initially, any track starting with either a DIB:S or a Snort observation gets a 0.3 score from this model. Then, as more DIB:S or Snort observations arrive the track score grows as follows:

$$S_{\mathcal{M}}(T_n) = \begin{cases} \frac{1}{2} \times (0.8 + S_{\mathcal{M}}(T_{n-1})) & : \text{ when source IPs match} \\ 0 & : \text{ otherwise} \end{cases}$$

An interesting artifact of this particular model is that our experimental worms initially show up as a list of RAT deploying hosts. This is understandable considering the aggressive scanning nature of worms, and the fact that they, after succesful infection, themselves start to scan and deploy the worm code. Because of this behavior the model tends to quickly collect many observations and thus starve other tracks from potentially useful observations. Therefore the best results with this model are achieved when it runs all by itself in a separate instance of a PQS. The concluding results from this PQS can then again be used as input to a tier-2 PQS tracker as evidence of "poorly behaved hosts".

Low&Slow Scans

sensors: IPtables
DIB:S
Snort

This model checks for one particular type of low and slow scanning behavior; one IP address scanning a large network, or multiple networks, on the same port using the same protocol. Two things are very important for this model; First, things go slow. This means that it often cannot be combined with other models in the same PQS, because pruning rates typical for other models will destroy most low-and-slow tracks before they even get a change to grow to a significant score. Second, because of the slow pruning configuration, a lot of very small tracks are formed and kept around with a very low score. Basically, any observation that may even remotely indicate scanning behavior is included by this model and forms the beginning of a new track. Then, only when more observations come in that show that a particular host is scanning slowly, the track grows and bubbles up to the top of the track list, where a conclusion can be drawn. For larger networks, this model can be a CPU and memory nightmare. As with the RAT model, the output of this model often better serves as input to higher level trackers.

For this model several ports were defined as *Extra Caution*; port/protocol combinations that should be watched in particular for the monitored network:

```
UDP 53       (DNS)
TCP 21       (FTP)
TCP 22       (SSH)
TCP 135-139  (NetBios)
TCP 27374    (SubSeven)
TCP 31337    (Backorifice)
```

Any track starting with a DIB:S, IPtables, or Snort observation will get an initial score of 0.1, or 0.3 if the observation was flagged as *Extra Caution*. When source IP, protocol, and destination ports are matched the score is grown as follows (and set to 0 otherwise):

$$S_{\mathcal{M}}(T_n) = \begin{cases} \frac{1}{4} \times (1.0 + 3 \times S_{\mathcal{M}}(T_{n-1})) & : \text{ normally} \\ \frac{1}{2} \times (1.0 + S_{\mathcal{M}}(T_{n-1})) & : \text{ for } \textit{Extra Caution} \end{cases}$$

Similar to previous models, the previous score assigned to the track by this model is taken into consideration in every score calculation. Since all tracks start off with a score of 0.1 (or 0.3, depending on service port) this model makes scores grow to 1.0, but only if destination ports match. If the service port requires *Extra Caution* the previously assigned score is weighted only one time to ensure that the track score grows faster as more evidence comes in. Otherwise the previous score is weighted three times.

Unauthorized Insider Document Access

sensors: SaMBa
Spade

This model focusses primarily on users accessing honeytokens on the central fileshare. It monitors all accesses to the fileshare using the SaMBa sensor, and checks it agains a list of known honeytokens. To improve performance this model also checks for

fileshare packets that have a high anomaly score, for example, when the fileshare is accessed at an unusual time of day. Initial scores are 0.9 if a honeytoken was accessed, 0.6 for any packet with a very high anomaly score going to the fileserver, 0.4 for any fileshare access violation, and 0.0 otherwise. Tracks scores change according to this function:

$$S_{\mathcal{M}}(T_n) = \begin{cases} \frac{1}{2} \times (1.0 + S_{\mathcal{M}}(T_{n-1})) & : \text{honeytoken access} \\ \frac{1}{2} \times (0.5 + S_{\mathcal{M}}(T_{n-1})) & : \text{general access violation} \\ \frac{1}{2} \times (1.0 + S_{\mathcal{M}}(T_{n-1})) & : \text{anomalous traffic to fileshare} \\ S_{\mathcal{M}}(T_{n-1}) & : \text{otherwise} \end{cases}$$

where in all cases either the SMB username or the source IP address of the request must be the same for all observations in the track.

4.4.2 Failures

This category of models was specifically designed to catch cases where the observed events are more likely associated with a network device failure than an attack. However, it is not always clear which of the two may be going on, for instance, consider a DDOS attack on a large network. The border router of this large network is likely to fail under such a heavy load, and therefore it is more likely that the system reports router failure than DDOS attack. (Either way there is only one thing the administrator can do: restart the router.)

Router or Link Failure

sensors: DIB:S
Snort

Technically the models for router failure and link failure are separated, however, it seems very difficult to separate the two conditions based on the observations we have available. Router failure happens when a router, somewhere upstream, fails to route packets. In some cases traffic will be dynamically re-routed, although usually network connectivity simply stops. Link failures are all conditions where the physical link failes, this can be a cut in the wire or a switch or hub failure. Both conditions are very similar and often require a lot more than network based sensors to diagnose the condition accurately.

The router failure model depends on DIB:S observations with a local source address, giving a score of 0.6 to all new tracks that fit this condition. More DIB:S observations with a local source address quickly let the total score grow to 1 according to $S_{\mathcal{M}}(T_n) = \frac{1}{2} \times (1.0 + S_{\mathcal{M}}(T_{n-1}))$. The drawback of this model is that it requires a local network installation of the DIB:S system, and at least one functioning router. If the local network router fails, then this model will be unable to detect the condition for any host on that local network. It will, however, work well for a larger enterprise class network that has multiple smaller (say class-C) networks, all monitored by DIB:S enabled routers. If any

of these routers for a class-C subnet fails, other hosts in the larger network will be unable to reach this small subnet. The ICMP-T3 traffic coming from the other routers going to the DIB:S system will then quickly generate alerts that will trigger this model effectively.

The link failure model relies on Snort observations and starts out with the premise that only the local subnet is still reachable. Therefore all other network based and global sensors are unreachable. The Snort observations are assumed to be originating from a local sensor, and are only considered for this model when their source address is within the local network. Only two rules are important to this model: Snort rule 472 (ICMP redirect host), and Snort rule 394 (ICMP-T3 destination unreachable). Both may (but do not necessarily have to) be generated by a reachable router closeby. When the source IP address on any of these two specific Snort observations is from the local network then the observation may grow the score in the track. The score starts out at 0.2 and grows to 0.9 when the destination IP address is the same for all observations, otherwise to 0.7, according to the following formula:

$$S_{\mathcal{M}}(T_n) = \begin{cases} \frac{1}{2} \times (0.9 + S_{\mathcal{M}}(T_{n-1})) & : \text{ iff destination IP is the same} \\ \frac{1}{2} \times (0.7 + S_{\mathcal{M}}(T_{n-1})) & : \text{ otherwise} \end{cases}$$

where the scores never really grow to 1.0, mostly because this model can never be really certain of what is going on. It may merely be the "best" explanation of what is happening, due to the lack of a better hypothesis.

DNS server misconfiguration

sensors: IPtables

This is a very straightforward model that simply checks if external systems are properly contacting the local DNS server. If name resolution requests consistently arrive at the wrong system in the network, then the DNS record may be wrong. This is a very rare condition that is not expected to occur frequently, but can be hard to diagnose from within the local network. The Netfilter firewall is configured to report on any UDP 53 (DNS) requests *not* going to the local DNS server. The model only scores tracks consisting of IPtables observations originating from outside the network, going to inside the network, with a destination other than the DNS server. Tracks start with a score of 0.3 and grow to 0.9 according to $S_{\mathcal{M}}(T_n) = \frac{1}{2} \times (0.9 + S_{\mathcal{M}}(T_{n-1}))$, given that all observations are UDP 53 DNS requests.

Email server breakdown

sensors: DIB:S

Since DIB:S reports on consistent groupings of destination unreachable messages, it can also be used to diagnose that critical hosts are not responding. This model gives a simple example on how these observations can be used in a PQS to detect that a specific mission critical system has failed. When many systems are all trying to reach the SMTP mail server, but the router is not getting a response on its ARP requests, then it will start returning ICMP-T3 messages. If these messages are processed by DIB:S, it will soon

issue an observation reporting that many systems attempted to contact machine: *mail* on port 25 using protocol TCP, and failed to do so. This model will therefore only accept DIB:S messages that are formatted exactly like that: destination must be *mail*, on TCP 25 (SMTP). Other protocols can be easily included, however. Tracks start with a score of 0.3 and quickly grow to 1.0 given $S_{\mathcal{M}}(T_n) = \frac{1}{2} \times (1.0 + S_{\mathcal{M}}(T_{n-1}))$.

4.4.3 Tier-2 Models

Tier-2 models refers to a category of models that have both sensor observations, as well as the output of other models available for evaluation. This means that the output of the models discussed so far may be used as input to these Tier-2 models. The output of the above models includes: track score, model name, source and destination IP (where appropriate), and source and destination port and protocol (where appropriate). These higher tier models are often very interesting because they deal with complex behavior. For instance, the two models discussed in this section are designed to track the higher level behavior of single hosts (host-state), and to track the progression of a multistage attack through a network (attack-state).

Consider Figure 4.3 where a Tier-2 PQS tracker is added to the architecture. Outputs from the Tier-1 tracker are used as input observations to the Tier-2 tracker, together with sensor observations. In practice there is often no particular need for a second PQS core, instead it is possible to loop the output from one core back into its own input, effectively combining both Tier-1 and Tier-2 models in the same tracking engine.

Based on Tier-1 output, or direct sensor observations the status of an individual host can be tracked. Consider the model in Figure 4.4 where a host is in one of seven states, three of which are considered "trusted", and the others are "hostile". Once a host is in a hostile state it cannot move back to a trusted state automatically. Notice how the "compromised" state is essentially a placeholder, waiting for further information on what activity the host will be undertaking, while hostile. The depiction of this model omits the specific sensor observations that are associated with each state, or state transition.

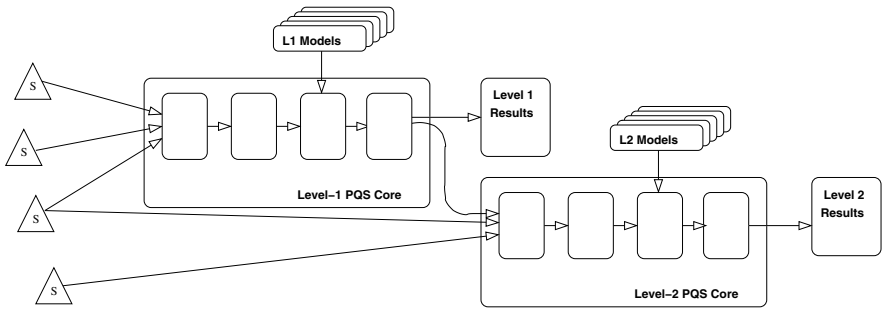


Figure 4.3: Toplevel view of a Tier-1 and a Tier-2 PQS tracker.

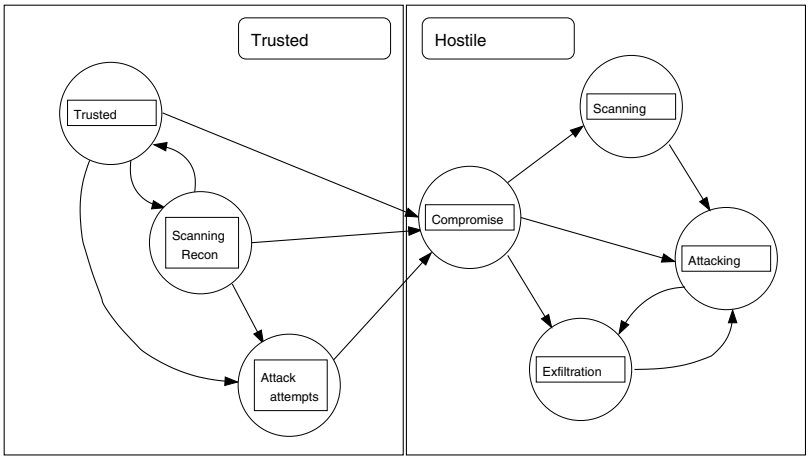


Figure 4.4: State machine for an individual host.

However, assigning these observations is intuitive, for instance, examples of "recon" are the results of the *low&slow* scan model, a DIB:S observation, or a Snort scan alert. This model also shows that there is not necessarily a structured set of state transitions that put a host from fully trusted into a compromised state; depending on received observations it is very well possible that a host is considered compromised without going through the reconnaisance or attack attempt states.

Figure 4.5 shows another Tier-2 model that tracks the progression of a complicated, multistage attack. The challenge here is to correlate the activities of various hosts together and identify which steps are related. Consider the following two Tier-1 observation sequences:

A	scans	**B**
A	attacks (failed)	**B**
A	scans	**C**
A	attacks (successfully)	**C**

and:

A	scans	**B**
C	attacks (failed)	**B**
A	scans	**B**
C	attacks (successfully)	**B**

In the first case it is clear that host A is being very hostile, and there will be little doubt that all four steps are related and part of the same multistage attack. It is unlikely that A's attack on B is unrelated to its attack on C, although both hypotheses will be generated by the PQS. It is expected that the first hypothesis will ultimately dominate because all steps originated from the same aggressor.

In the second sequence, however, this is not as clear cut. Although host A is doing

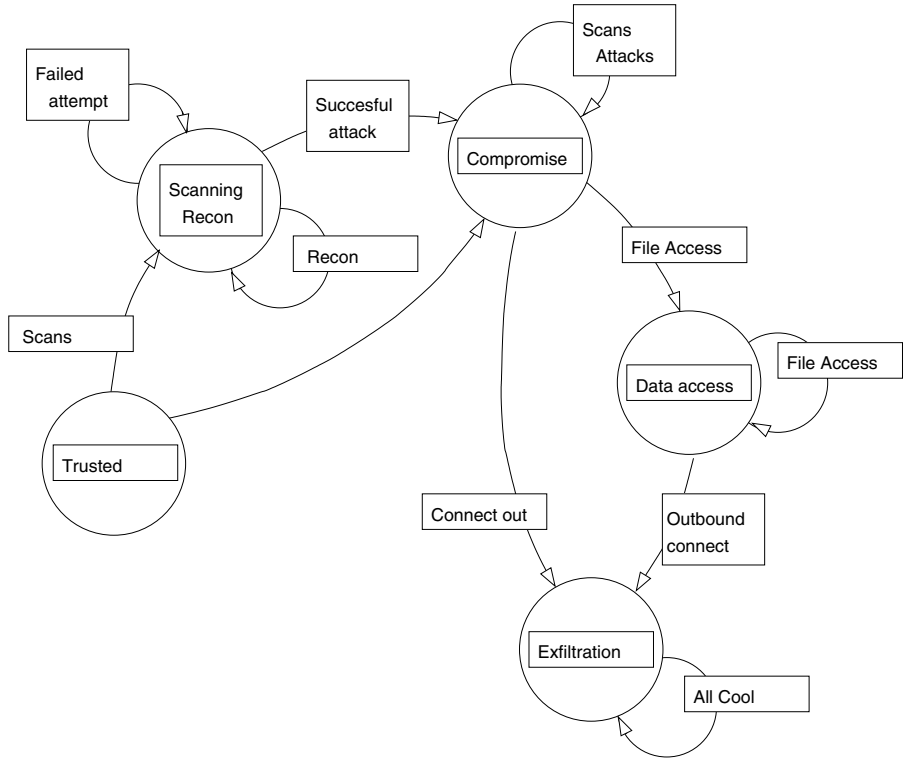


Figure 4.5: Level-2 model tracking steps in a multistage attack.

the scanning, it is host C that actually performs the attacks. The PQS will generate two hypotheses (one assuming that all steps are related, the other assuming that the scanning is independent from the attacks), each of which has a separate set of reasons why it is a likely hypothesis. For example, it could be argued that the target is B, and that the attacker is using multiple systems to attempt a compromise. On the other hand A and C are separate machines acting independently. If, later on, evidence comes in that A is actually scanning the entire subnet, whereas C has only attempted to attack one system, then it becomes more likely that their actions are, in fact, independent, although the other scenario cannot be ruled out. In the end, however, the human analyst will probably care more about the fact that B was compromised than whether A and C are controlled by one and the same aggressor. Finally, consider the following observation sequence reported by the Tier-1 PQS:

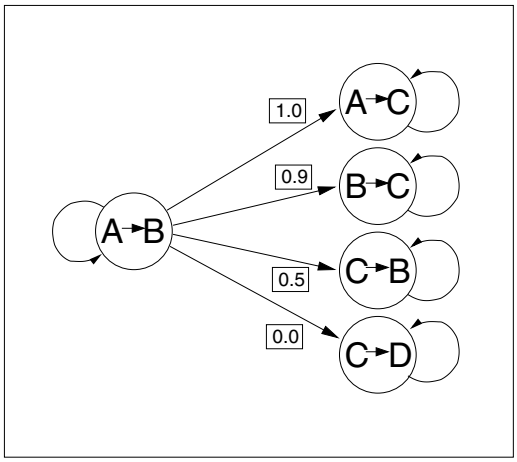


Figure 4.6:

A	attacks	**D**
B	attacks	**D**
C	attacks	**D**
D	scans	**E**
D	attacks	**E**

Although it is obvious that host D is successfully used as a stepping stone, it is not clear which of the hosts A, B, or C initially successfully compromised D. Initially all three will be part of the hypothesis-set, however, as time passes and more evidence arrives a definitive determination may be generated. In other cases, the true chain of attacks may never become fully clear.

Figure 4.6 gives a graphical representation of this multistage attack model. The scores are calculated as follows, based on the two IP addresses X and Y in the new observation:

$$S_{\mathcal{M}}(T_n) = \begin{cases} \frac{1}{2} \times (N + S_{\mathcal{M}}(T_{n-1})) & : \text{ iff either } X \text{ or } Y \text{ occur in track} \\ 0 & : \text{ otherwise} \end{cases}$$

where Figure 4.6 gives score N. The model is actually matched backward, searching from the *newest* observation back into the track. Self-transitions do not impact the score, and are consumed until one of the four actual transitions is found. At that point the score N is assigned. IP addresses X and Y are therefore matched to the two IP addresses in the matching final state.

4.5 Results and Lessons Learned

Process Query System technology is still maturing, however its use as a powerful and flexible data processing system has been proven by various applications. The goal of the PQS-Net system was to build a system that can assist intrusion analysts, not replace them. Data is conveniently collected from all over the network through the PQS sensor interface and evaluated by one or multiple models. So far the models have reflected relatively straightforward relationships between events, however, these relationships are the same as the associations made by analysts. Based on these associations the analyst will often search for more information or evidence by collecting logs, watching traffic, or reading websites on global Internet trends. Collecting data is probably the single most time consuming task for any analyst. The PQS-Net system effectively eliminates the need for the human to go get the data, look at it, and decide what other information to look for.

4.5.1 Performance

Since all of the basic network security sensors are present (IDS monitoring, firewall logs, etc.) the PQS-Net system does not miss any intrusions or attacks if they are picked up by the sensors. Therefore, accuracy should not be measured as a factor of the false positives and false negatives, as this would only directly reflect the accuracy of the sensors, however, a more meaningful measure is to evaluate if the associations made by the models give an accurate top-down list of important relationships. In other words, whether the system is properly telling the analyst what is "important", and what is "unimportant" in the flow of observations from the sensors. The problem is that "importance" is a very subjective term because it depends not only on network data, but also on the specific purpose of the systems, and the value placed on them by the organization. For example, a bank will place a very high value on the system that holds the account information. When the bank gets attacked, the analyst will like to see an intrusion of the account computer at the top of the list, preferably not cluttered with alerts about scanned teller workstations. This is what the PQS-Net system does.

So, in terms of accuracy, a better measure of performance would be to look at the number of correct associations made. To test the system structurally, several large attack datasets were sent through the PQS-Net system at full speed. Each of these datasets represented networks of roughly 1260 hosts, dozens of which were public servers. The attack scenarios included multiple concurrently acting aggressors (usually 3 or 4 attacks going on at once), decoy scanning hosts (not actually associated with the attacks), several victims and stepping stones (usually server hosts that got compromised and used to gain deeper access into the network). Furthermore, the datasets contained many Gigabytes of normal, benign traffic. Now, based on all the observed traffic and log files, how many of the attackers, decoys, victims, and stepping stones were correctly correlated and reported to the administrator? The results are presented in Table 4.1.

The results in this table represent a significant data reduction from tens of thousands

Dataset:	Attackers	Decoys	Victims	Stepping stones
1	3 of 3 (100%)	5 of 5 (100%)	2 of 2 (100%)	1 of 1 (100%)
2	4 of 4 (100%)	2 of 2 (100%)	2 of 2 (100%)	1 of 1 (100%)
3	0 of 2 (0%)	2 of 2 (100%)	1 of 2 (50%)	1 of 2 (50%)
4	3 of 5 (60%)	6 of 6 (100%)	10 of 11 (91%)	2 of 3 (67%)
total	10 of 14 (71%)	15 of 15 (100%)	15 of 17 (88%)	5 of 7 (71%)

Table 4.1: PQS-Net performance on 4 blind datasets. The operators of the system were unaware of which attacks would take place.

of alerts to only several. For example, the first dataset contained 22,930 alerts (total count from all sensors), which the PQS-Net system boiled down to 100 tracks (or events), that the administrator should take note of. For the fourth dataset the reduction was even bigger, from 39270 alerts to only 107 tracks (a reduction of 1:367).

In terms of efficiency we can only give some rough numbers on resources used. The first dataset contained 1.5 Gigabytes of traffic, covering a real-time duration of exactly 47 minutes. A total of 62 sensors were used in this test. A 1Ghz Ultra SPARC III system ran this dataset through using about 80% of the available CPU time, which means it can handle a network of about 1500 hosts maximum. The memory requirements, however, were minimal: under 100 MB total. Note, however, that no official studies were done to evaluate if human analysts showed improved productivity using the PQS-Net system, although several analysts responded, when asked for their opinions, with: "This really rocks!", "Wahooo", "Great work", "When can we start using this?", and "Thanks mate!".

4.5.2 Considerations

Regarding ease of applying a PQS to any particular area a couple of things can be said:

- Model building needs to be simplified. At this point most models are written directly in Java code, PQML or XML. It is quite straightforward to envision graphical user interfaces that will make model building significantly easier. Especially considering the frequent re-use of old code in new models, these GUIs will compile the models into PQML, which is the portable modeling language for trackers. Since PQML is an assembly-inspired language, it suffers from the same flaw that assembly languages suffer from: not everyone is comfortable writing it. Improved user interfaces can therefore make a big difference.

- More models in a single tracker, or each model in its own tracker. This is one of the bigger question marks of the current PQS-Net system. It seems that some models (low&slow for instance) require their own tracker because of specific tuning parameters that make the model work. However, in other cases the system performs better when the models compete for observations in the same tracker, thus forming many meaningful hypotheses.

- How different are two models? Closely related with the previous point is the question of how *different* two models are. Although multiple models were constructed for detecting worms, they all seem to catch the same worms equally well, based on very different associations. This means that simply looking at the models and deciding that their associations are "very different" or "very similar" says nothing about their expected tracking accuracy. However, it is important to have some measure of similarity. Consider, for example, a case where two models are built to detect two similar events in a noisy environment. If the models are "very similar" it may actually mean that there is too little sensor information available for the models to make a clear distinction between the events. When another type of sensor source is added the models may be able to do a better job simply because more information is provided.

- When to draw *hard conclusions*. Although the PQS-Net system was designed to assist analysts by showing event correlations in order of importance, it can be argued that in some cases an autonomic response would be desirable. In order for that to work the system must be able to draw hard conclusions and act on them. Hard conclusions are usually drawn when the score of a track goes above 0.8, or 0.9. If there are multiple competing models all claiming a different conclusion, then the score will have to be closer to 1 before a conclusion can be drawn. The tuning of this parameter will depend highly on the specific circumstances.

The above considerations will drive development of the PQS-Net system for awhile to come. As mentioned in the last point it is often difficult to accurately present an administrator with *useful* input, without being overwhelming. In computer network security, enormous quantities of data are available from all over the network. The PQS therefore becomes a data reduction filter, looking for the most important signs, however, also to a human analyst it is often not clear what these signs are. Thus model building will only be as good as the analyst supplying the expert knowledge for the models.

Chapter 5

Other Applications

In this chapter three applications of PQS are introduced that were developed based on the TRAFEN platform. They demonstrate the wide applicability of the PQS concept. The first application is a detector for *timing covert channels*, which are a highly stealthy way of exfiltrating data. By perturbing the time of release of packets in a benign communication, the attacker encodes the desired information. The models focus on identifying the statistical properties of these covert channels. The second application tracks fish swimming in a tank, giving a good, real-world example of kinematic tracking with a PQS. The tank is filmed by a camera, and X,Y-coordinates are generated of anything that moves. A PQS is then used with a relatively simple model to track the fish as they move around and behind obstacles. It is a very intuitive way of presenting the PQS concept. Finally, for the third application, TRAFEN is used to monitor servers in a network, and take autonomous action based on the detected processes. Examples of processes that are detected include the failure of a service, degredation of response times or system resources, and the effects of a succesful network intrusion.

5.1 Covert Channels

This section discusses a technology that was developed as an extension to the PQS-Net framework. It uses a PQS tracker of its own, and its output has been successfully used as observation inputs to the PQS-Net tier-2 trackers. This technology focuses on methods of detecting *Timing Covert Channels*, a technique where the interpacket timing of benign traffic is perturbed to encode information.

5.1.1 Introduction

The success of modern network defense and intrusion detection has forced hackers to be more and more creative with their attack methods. In a situation where an attacker has infiltrated a tightly monitored network with the goal of stealing information, he or she runs the risk of being detected when the information is exfiltrated.

Assuming the network is heavily guarded with Intrusion Detection Systems, Packet Anomaly Detection systems, and firewalls, the intruder has limited options on getting the stolen data out. The most straightforward method for exfiltrating data is the use of a conventional protocol, like FTP. This runs the risk of being detected in log-files and traffic dumps, and running a similar communication on irregular, high port numbers might even trigger Packet Anomaly Detection systems, because such communications are highly unusual. Encoding data in the unused space in packet headers is likely to set off most modern intrusion detection systems, as will packing data in the payload section of Ping packets. Additionally, transmitting data through a Ping payload might trigger Packet Anomaly Detection systems when the size of the Ping packets is increased and/or irregular. The attacker will thus have to look for even more covert ways of moving the data out of the compromised network.

The data exfiltration option under investigation in this section is the utilization of interpacket delay times to encode data. This means that the intruder does not necessarily have to create traffic, however, he or she can modulate the time between packets of regular communications to encode the data. The extent to which existing traffic streams can be used depends on the location of the recipient. The intruder will need access to the communication to measure the packet interarrival times, in order to retrieve the data.

We investigated two techniques for detecting possible use of interpacket delays for detection of covert channels. The first technique is based on the calculation of the entropy of the suspected bit sequence. The entropy here measures the amount of chaos in data, meaning that completely random data will have an entropy of 1.0, and thus not hold any information. However, when data does hold information it is structured and not fully irregular, thus having an entropy smaller then 1.0. We use a data compression technique to see if the suspected bitstream is compressible; if so, the data has regularities and the entropy is lower. The second technique simply tries to identify multiple concentrations of interpacket delay times. These concentrations should be sufficiently differentiable, in order for it to encode a binary bitstream.

5.1.2 Background on Covert Communication Channels

Since security analysts first started thinking about covert channel communication, two terms were introduced: *covert storage channels* and *timing covert channels*. Covert storage channels involve the writing to a storage location by one process and the reading of the storage location by another process. The resource, such as unused bits in a packet header, or the padding fields in a datagram, is shared between the two subjects. With timing covert channels the sender transmits data to the receiver by modulating its use of system resources in such a way that the manipulation affects the response time observed by the receiver. Specifically, this is done by modulating the wait time between sending packets (the interpacket delays). Since various techniques already exist to detect the first type of covert communication, the second type is the focus of our research.

The general form of a covert communications channel is based on the idea of exploiting time delays between transmitted packets in order to implement a form of Morse-like code. Intuitively this means that a short time delay between two consecutive packets encodes a binary zero, and a long time delay encodes a binary one. More generally, suppose an outside intruder has been able to gain control over a machine X inside our network and wishes to send data to his/her computer A by codifying the information as time delays between packets. (Note that A does not have to be the destination for the network packets, however, A merely needs to be on the path such that the packets may be intercepted and their interpacket delays measured.) For example we can imagine that the intruder is able to execute instruction *ping A* from the compromised machine X within the perimeter of our network.

- **ping A**
- **ping A**, after Δt_1 seconds

- **ping A**, after Δt_2 seconds
- **ping A**, after Δt_3 seconds
- ...

Machine *A* intercepts a sequence of pings with time delays between consecutive packets as Δt_1 seconds, Δt_2 seconds, Δt_3 seconds and so on. From an abstract point of view this technique amounts to sending symbols of a source through a noisy channel, and decoding the message at the destination. We consider the fact that the delays at the arrival computer are not exactly the same as in the departure computer due to noise within the Internet. Any forwarding device (routers, firewalls, switches, repeaters) will incur a small processing delay. This delay will vary in time, thus perturbing the interpacket times, making the channel noisy.

Assume that all the packets flowing through our network are filtered by an agent (e.g. a firewall) that can record packet interdelays for each related communication (for example: pings to the same destination address, or a stateful TCP session with a web server). From this agent we obtain a list of delays. The question we try to answer is: "Given a chain of consecutive delays Δt_i, is it possible to affirm with a certain probability that there has been malicious intent from somebody within our network?". In other words, "Are the observed delay times specific/discernable enough to (most likely) *not* have been generated by chance."

5.1.3 Assumptions and Considerations

Since the general problem is very broad we start with some considerations that give rise to several restrictive assumptions that will reduce the complexity of the situation.

The first difficulty in detecting a covert channel comes from the problem that we cannot know the cardinality of the encoding that the attacker is using. Meaning that it is not clear from the start if the attacker is using two, three, or *N* different delays. Intuitively, the attacker may pick an encoding that maximizes the bandwidth of the transmission. However, the decision to use a given number of symbols (or delays) is not always a matter of maximizing transmission bandwidth. Consider, for instance, that transmitting a given datafile though a given covert channel would maximize the bandwidth when 2 symbols (ie. 2 different delays) are used. Since it is often hard to control existing sessions (it usually requires root privileges), the attacker will be better off sending HTTP requests to a web server that he/she controls (alternatively, Ping could be used). By modulating the times between the requests the attacker will be able to encode the transmission.

Although the transmission rate is maximized in this case, it is also, by far the most noisy. For every bit transmitted an HTTP request is made, something that may stand out when the traffic on the network is analyzed. Conversely, consider that the attacker prefers stealth over transmission speed. Conceivably a 64 symbol encoding can be chosen, meaning that each HTTP request then carries 6 bits of information ($2^6 = 64$). It is therefore illogical to assume the attacker will pick a given encoding based on the maximizations of transmission speed. Especially when we consider that the attacker chooses

a covert channel to communicate, it is likely that he/she will try to maximize stealth over transmission bandwidth.

Additionally, for the aforementioned reasons the communication to modulate would preferably be a continuous datastream, for example, the acknowledgments of a real-time audio stream. An interactive session, such as SSH, will not have a predictable release of packets (ie. only when the user types a command) to allow for transmission of large files. Add to that the fact that the attacker will rarely know beforehand what communications will predictably exist; it is therefore easiest to simply create the traffic, instead of using existing connections. (Although this has an increased risk of unwanted detection.)

It must be noted, however, that an attacker will most likely not have the luxury of choosing the most efficient or most stealthy encoding possible, for it assumes that the attacker can have some form of information exchange with the compromised system. If no such bilateral communication channel exists beforehand, the attacker will never be able to instruct the attacking agent on what encoding is best received (and therefore preferable). For that reason we may safely assume that an attacker will pick an encoding that will yield a *decent* bandwidth on average, while being *sufficiently* stealthy. With this in mind, we continue with the following two restrictions.

1. *Malicious Noise*. The intruder does not add artificial noise with the purpose of preventing the observer from detecting his/her activity. Artificial noise can be added by wildly varying the interdelays at a predetermined moment. For example, the intruder resolves to send a sequence of packets with randomized interdelays after successful transmission of every 100 bits. The intruder knows to discard these random interdelays as intended noise, however, our algorithms do not, therefore the algorithms might fail to detect malicious activity in the blocks of 100 actual bits.

2. *Binary source*. The intruder implements a code based on two symbols. So there will be only two intervals of time Δt_1 and Δt_2 that codify two symbols 0 and 1 respectively, as in the classical Morse code. This assumption may seem very restrictive as the intruder could try to codify information based on more than two intervals of time and so more than two symbols. The detection methods discussed here are, however, easily extensible to handle other encodings as well.

5.1.4 Covert Channels and their Capacity

Before we can say anything useful about detecting covert channels we first investigate the properties of such a channel. We consider the error rates, and the channel capacity. To test our algorithm we implemented a version of the covert channel with a self-calibrating delay loop in the sender. This means that the sender automatically adjusts sleep times for operating system overhead and system load to ensure that the timing is accurate. An arbitrary string of bytes was sent and then verified at the receivers' end. Error rates were measured for zeroes being received as ones, and ones being received as zeroes.

Experiments were run where the receiver was 4 hops, and 24 hops away, however, the error rate for 4 hops was mostly dependent on system load instead of network latency

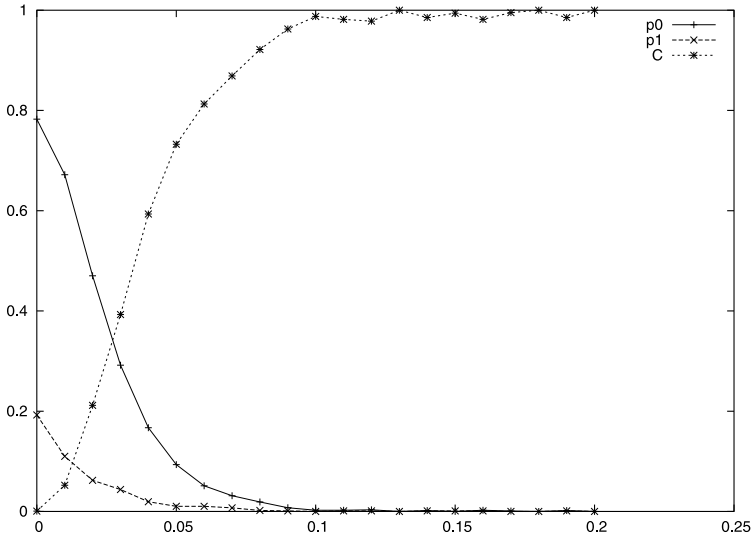


Figure 5.1: Error rates for zeroes being received as ones (p_0) and ones being received as zeroes (p_1), and channel capacity (C) for non-symmetric binary channel, where Δt_0 is constant and Δt_1 increases. Horizontal axis shows $\Delta t_1 - \Delta t_0$ in seconds, $\Delta t_0 = 0.2$s, $0.21 \leq \Delta t_1 \leq 0.4$s

differences. Therefore all 4 hops data was discarded. Over the course of multiple days various predetermined sequences were transmitted and the error rates were recorded. The difference between Δt_0 and Δt_1 was varied from 0.01 second to 0.2 seconds. The graph in Figure 5.1 shows the error rates p_0 and p_1 that were measured, where p_0 is the fraction of zeroes that were received as ones, and p_1 is the fraction of ones that were received as zeroes.

There is no error-free communication over a noisy channel when messages are encoded with zero redundancy. The amount of redundancy that must be added to achieve error-free communication depends on how noisy the channel is, and can be measured with the Shannon Channel Capacity [90]. The channel capacity represents the highest amount of information per symbol (Bit/Symbol) that can be transmitted through the given noisy channel, and for a binary symmetric channel (BSC) is defined as

$$C = 1 - \left[P_e \log \frac{1}{P_e} + (1 - P_e) \log \frac{1}{1 - P_e} \right] \tag{5.1}$$

where P_e is the error probability. For example, if we observe a train of time delays that form a "random" (incompressible) binary sequence then we would expect that the number of zeroes is about the same as the number of ones. However if $C < 1$ then it would be impossible for the intruder to convey any information (with a negligible error proba-

bility) transmitting at a rate of 1 Bit/Symbol. In other words, in the presence of noise, the intruder will in general be forced to adopt some form of channel codification with a rate necessarily below C. This redundancy is well known in almost any form of digital communication; popular schemes include sending parity bits, CRC (Cyclic Redundancy Check) codes, and ECC (Error Correcting Code). However, when the C falls far below 1, the amount of redundancy that the intruder is forced to use becomes impractical. This is also the solution to preventing this type of covert channel: by artificially varying packet delays at the border (be it router or firewall) enough to force C down so far that successful transmission of data will go too slow, and is too unreliable.

However, intuitively our binary channel is not symmetric. Since the interpacket delays are different for zeroes and ones, their respective transmission rates are different as well. This leads to the expectation that the error rates must be different also, and this can be observed from the graphs (p_0 and p_1 are different). In other words, it takes less time to transmit 100 zeroes than it takes to transmit 100 ones. This means that the rate at which zeroes are transmitted is *higher* than the rate at which ones are transmitted. (The fact that zeroes and ones are intermixed in our communication does not matter; the results would have been the same where all the zeroes transmitted first, and then all the ones.) Assuming that the channel does not care whether zeroes or ones are transmitted, then the error rate is expected to go up as the transmission rate goes up. This explains why the error rate p_0 (zeroes being incorrectly received as ones) is much higher than the error rate p_1 (ones being received as zeroes). As the difference $\Delta t_1 - \Delta t_0$ increases the error curves both go down, since it becomes easier to distinguish between zeroes and ones. Given error rates p_0 and p_1 the channel capacity for a non-symmetric binary channel becomes [81]:

$$C = \log\left(1 + 2^{\frac{H(p_0)-H(p_1)}{p_1+p_0-1}}\right) + \frac{(1-p_0)H(p_1) - p_1 H(p_0)}{p_1 + p_0 - 1} \tag{5.2}$$

where

$$H(x) = -x\log x - (1-x)\log(1-x) \tag{5.3}$$

The graph in Figure 5.1 shows the channel capacity for a non-symmetric binary channel, based on the measured error probabilities. Finally some observations about the experiments need to be made explicit. Note that the time of day and network load can seriously affect the accuracy of the transmission; this means that a congested network forces the sender to adopt larger differences between Δt_0 and Δt_1, thus bringing down transmission speed. Secondly, the path that the packets traverse is of direct impact on the differences in interpacket latencies; not all 24 hops anywhere on the Internet will give the same graph. Larger backbones tend to have faster switching hardware, keeping differences in latencies to a minimum. We found that crossing oceans has the most profound impact on transmission speeds, sometimes lowering them by a factor of 10. Likewise, rate limiting and quality-of-service queueing may at times completely distort the channel, while at other times allowing flawless transmission. This said, it must be realized that the graph in Figure 5.1 teaches us more about the expected shape and best-case scenario, than it tells us about what can be expected in general. The observed differences between Δt_0

and Δt_1 (horizontal axis) would certainly incur much higher error rates using different Internet paths.

5.1.5 Detection techniques

In this section we propose two methods for detecting a covert channel in interpacket delays, measured at an arbitrary point in the network. Both techniques are based on capturing related datastreams, meaning either a TCP communication, a PING to the same host address or network, or a series of UDP packets with the same destination host and port. The observation sequence that is extracted consists of the delays between *outgoing* packets. Thus each time a packet leaves the local network, the Δt is reported since the last packet went out to the same destination. Any returning acknowledgments hold no value for an outgoing covert channel. A PQS was used with a simple model to associate packets with streams, and since this is a very straightforward task, the data processing capabilities of a PQS allowed us to create this application very quickly. The PQML model produces histograms of interpacket delay times for a configurable number of seconds, for each tracked network communication. Below is the pseudo C-code for this model:

```
double Score(Track *t, Obs *o)
{
    Obs *last = t -> getLast();

    if (last -> src_ip != o -> src_ip ||
        last -> dst_ip != o -> dst_ip ||
        last -> proto  != o -> proto  ||
        last -> sport  != o -> sport  ||
        last -> dport  != o -> dport)
      return(0.0);

    int i = floor ((o -> ts - last -> ts) * 10)
    t -> hist[i] += 1;

    double score = t -> length / (t -> length + 10);
    return(score);
}
```

Since a detector is unaware of the time encoding that the attacker is using, an assumption must be made to attempt differentiation between the Δt's. In our system all the delays are stored and an average is calculated every time a new delay comes in. This average delay (basically the sample mean) is used as a cutoff to designate the delays as either zeroes (when they are smaller than the mean) or ones (when they are larger than the mean). This estimated bit sequence is re-evaluated for each new delay that comes in, and therefore can change as the mean is adjusted.

Detection by measuring Entropy

The goal of this approach is to find if the estimated bit sequence holds information, or if it is completely random. We do this by measuring the entropy of the bit sequence. Entropy is the measure of chaos, meaning that a higher entropy indicates that there is less order in the data. Completely random data will give an entropy of 1.0, the highest possible. However, when information is transmitted the data is structured and will therefore have a lower entropy. Measuring entropy is difficult and subjective; however, a good approach is to measure how well the data can be compressed. Compression (like Huffman, or Lempel-Ziv encoding) identifies repetitions and regularities in data and codifies recurrences with shorter codes, thus saving space. This means that the more regular the data is, the smaller it can be compressed. For entropy we give:

$$h_t = \frac{LZ(O^L)}{L} \tag{5.4}$$

where h_t represents the entropy, O^L is the bit sequence of length L, and $LZ(O^L)$ is the length of the compressed bit sequence. A covert channel is detected when h_t drops below 0.95 or 0.90. Figure 5.2 shows the entropy for (a) normal traffic, and (b) a sentence transmitted over a binary covert channel. The horizontal axis holds the length of the bit sequence. It can easily be seen that the compressibility of the packet interarrival times of regular network traffic is very poor, keeping the entropy at 1.0. However the compressibility of a covert communication does not become apparent until a large sequence has already been sent. Although the entropy drops below 1.0, it does not do so until at least 3000 packets have been transmitted. Although this technique works, it might not be practical (also note the scale on the vertical axis: the entropy does not drop far below 1.0). Additionally, once a chunk of data is compressed the regularities are exploited and the sequence will have an entropy of 1.0. This means that when the attacker decides to compress his/her communication this technique fails altogether.

Statistical Detection

This method is based on the assumption that (for a binary channel) the interpacket delays will center around two distinct values (ie. two distinct delays) while in a normal communication where the delays are more or less random, the Δt's are spread around a single spike, see Figure 5.3. We take advantage of this characteristic to make some statistical comparisons between a covert communication channel and normal network traffic.

Looking at Figure 5.3 the difference between a binary covert channel (graph *(b)*) and regular traffic (graph *(a)*) becomes evident; two spikes versus one big spike. For the covert channel the sample mean μ (average) of the interpacket delays will be somewhere between the two spikes. The packet count in the histogram at that point will therefore be very low. However, looking at a normal traffic pattern the mean of the interpacket delays will be in the center of the large spike. The packet count at the mean will thus be very high, if not the highest. If we compare the packet count at the mean by the maximum packet count from the histogram, we get a measure of how likely it is that the communication is a covert channel:

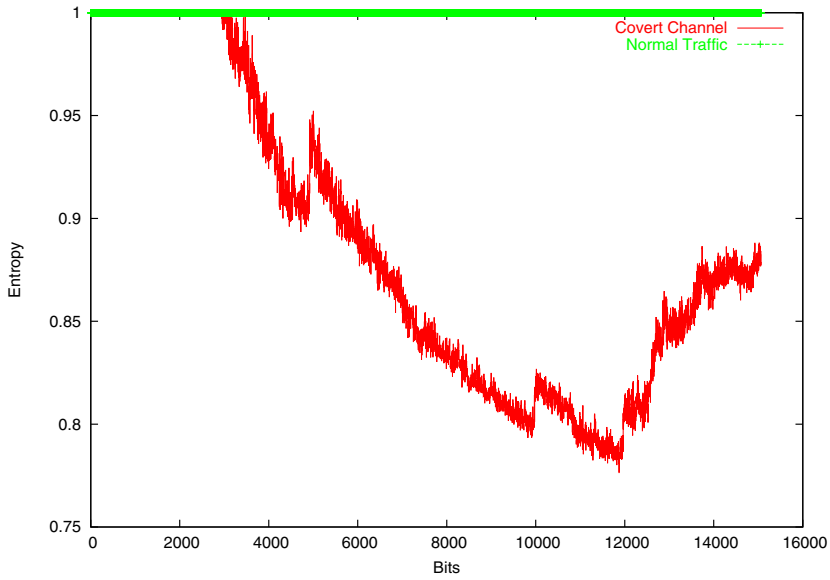


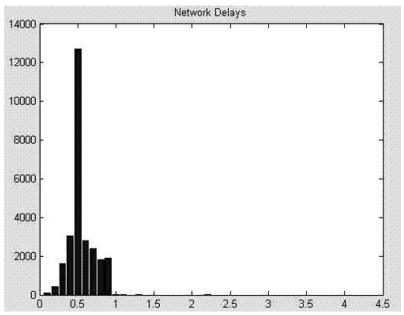
Figure 5.2: Entropy (compressibility) of bit sequence estimated based on interpacket delay times of (a) normal traffic, and (b) sentence transmitted over binary cover channel. Horizontal axis shows the length of the estimated sequence in bits, vertical axis shows the entropy.

$$P_{CovChan} = 1 - \frac{C(\mu)}{C_{max}} \tag{5.5}$$

Experiments with three different types of data were conducted, and Figure 5.4 shows the ratio $\frac{C(\mu)}{C_{max}}$ for these experiments.

1. *Normal Data.* Packets with a average delay of 0.2 seconds were transmitted. The interarrival times vary but the spike is at 0.2 seconds. The sample mean μ is therefore represented by a delay very close to 0.2 and the number of packets with exactly that delay ($C(\mu)$) is very high. The ratio between this number and the maximum number of a packet for a certain delay (C_{max}) quickly grows to 1.0, and stays there as more packets are transmitted. Normal traffic is very bursty and interpacket delays are often dependent on how quickly acknowledgments or responses are returned, which is once again dependent on the distance (and system load) of the two systems communicating. The packet delays therefore center mostly around a single value (0.2 seconds in this case) with occasional outliers (usually in the order or 120 seconds or more).

2. *Random Data.* Packets are sent with a fully random delay. Although this is not realistic for traffic encountered on the network, it does present a good idea of the

(a) 

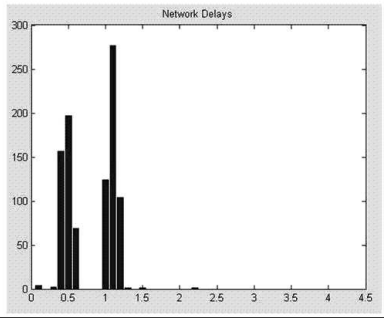
(b) 

Figure 5.3: Histograms for two different network communications. Horizontal axis shows the interpacket delay time in tenths of a second, vertical axis shows the number of packets received with a given delay. *(a)* example HTTP traffic session captured at our border. *(b)* a covert channel using UDP traffic, receiver was 24 hops away.

worst case scenario. Initially, when only a few bits have been sent, the delays scatter across the range and it is unlikely that the sample mean will have a high count. That explains why, until approximately the first 10 bytes, the ratio $\frac{C(\mu)}{C_{max}}$ remains zero. Later on, as more packets arrive the histogram starts to even out so the ratio starts to crawl up. As the number of transmitted packets goes up even further the ratio keeps growing, until it eventually hits 1.0 as the packet count goes to infinity.

3. *Covert Channel Communication.* Two delays are used, thus the interarrival times concentrate around those two values. The sample mean μ lies approximately in the middle between the two spikes. The count $C(\mu)$ is low and therefore the ratio $\frac{C(\mu)}{C_{max}}$ is approximately zero. As more and more bits are transmitted over the covert channel the spikes increase in size, however that ratio always remains very close to zero.

Our algorithm detects the sequence that most likely represents the covert communication channel analyzing the value $\frac{C(\mu)}{C_{max}}$. The lower that value the higher is the probability of having a malicious communication hidden in interpacket delays.

5.1.6 Discussion

The statistical analysis of the interpacket delays seems to do the best job of classifying between regular network traffic and traffic that is communicating through modulation of the interpacket delays. Therefore the covert channel detection models were all based on the statistical technique. As outlined above the actual sensor also runs inside a PQS,

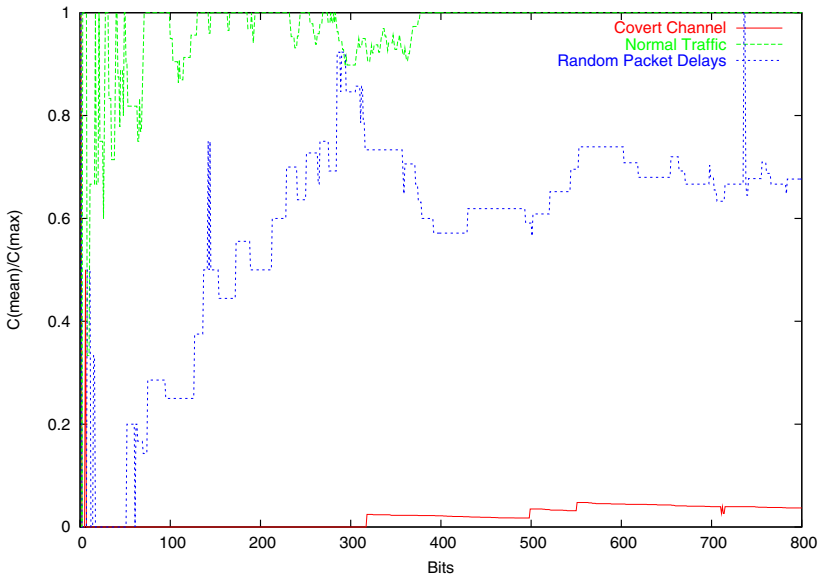


Figure 5.4: The ratio between the packet count at the sample mean and the maximum packet count for normal traffic, fully random delays, and for a binary covert channel. Horizontal axis shows the length of the estimated sequence in bits, vertical axis shows $\frac{C(\mu)}{C_{max}}$.

generating histograms of packet delays for all observed traffic streams. The PQS detection model takes these histograms as input and compares it to previously seen histograms of the same traffic stream, effectively smoothing out the outliers. The second step is evaluating the number of packets that were transmitted in the monitored stream; after all, if only 17 packets were transmitted so far the attacker could only have exfiltrated 2 bytes. This is hardly worthy of a red alert. Thus, as the number of packets transmitted grows, and the model still considers the histograms evidence of a covert channel, then the score of this alert is increased. We are currently investigating a method of further improving this model by comparing the histograms of a given network communication with previously recorded histograms of the same traffic type. This will refine the model by suppressing false positives, as the histogram of an interactive SSH session will look significantly different from the histograms of a HTTP web page download.

Finally, it is important to point out the intuitive relationship between the relative height of C_μ and the channel capacity. Consider that, for an approximately equal distribution of input symbols, C_μ forms the cutoff point between zeroes and ones; if a received Δt is smaller than T_μ a zero is received, otherwise a one. Therefore, in a histogram, the more packets are counted exactly around the mean (C_μ), the "higher" the confusion. In other words: the peak for zeroes and the peak for ones are overlapping. The higher C_μ, the larger the overlap. Confusion, or overlap, means that zeroes are incorrectly received

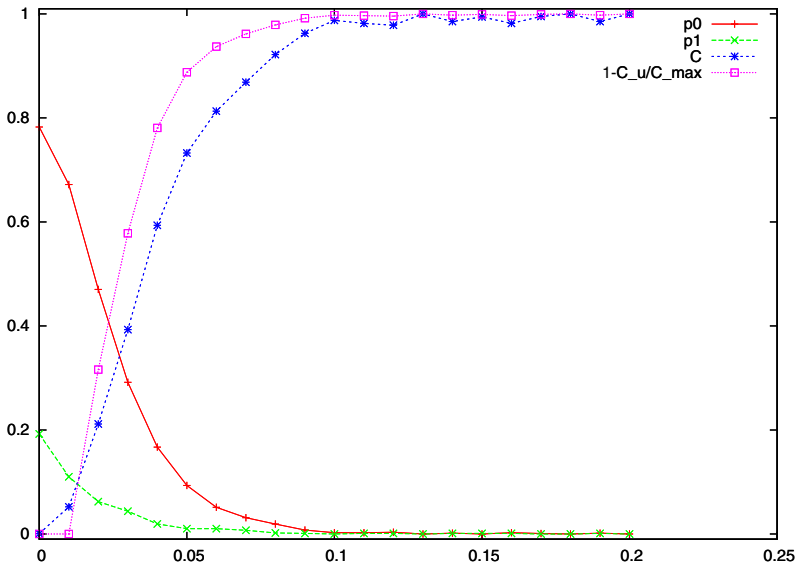


Figure 5.5: Same graph as 5.1 with the channel capacity added. Note the close relationship between the channel capacity C and the ratio $1 - \frac{C_\mu}{C_{max}}$.

as ones, and vice versa. A bigger overlap means higher error rates and a reduced channel capacity. So as the ratio $\frac{C_\mu}{C_{max}}$ grows, the channel capacity drops.

Furthermore, we know that when the peaks are perfectly separated the error will be zero, and therefore the channel capacity one. Also $1 - \frac{C_\mu}{C_{max}}$ will be one. Conversely, when the peaks are exactly overlapping $1 - \frac{C_\mu}{C_{max}}$ will be zero, and there will be no distinction between zeroes and ones, meaning that the channel capacity will be zero also. The intuitive expectation that $1 - \frac{C_\mu}{C_{max}}$ and the channel capacity are closely related is confirmed by the experiments shown in Figure 5.5.

5.2 Kinematic Tracking

This section describes the use of a PQS to track the motion of fish in a fish-tank, using a video camera. The PQS models used are very similar to the model presented with the dot tracking example in Section 2.5. This work was done together with Alex Jordan, who presented a Masters thesis on the subject [57]. An in-depth description of the system can be found there.

5.2.1 Introduction

Many tracking problems are centered around the kinematic properties of the object to track. Examples include tracking of airplanes, vehicles moving through a battlefield, people walking through the streets, the balls that were used in section 2.6, and tracking fish in a tank. In all these examples there are multiple objects that have a constantly changing position and momentum. The fish are particularly interesting since their changes in momentum are very sudden (fish tend to sit still for awhile and then quickly move and turn before settling down again). The goal of this project was to use streaming video images of the fish, track them, and predict where they are going in real-time.

5.2.2 Design

Sensors

The fish-tank was filmed by a low resolution camera providing a steady stream of frames to an image processing library that extracted "centroids" of anything that was bright in comparison with the rest of the frame. This means that colored fish, air bubbles, and moving plants stand out and get processed to an (X,Y) coordinate pair indicating the center of the object. The sensor therefore includes the camera and the processing that yields the centroids. The output of the sensor is a stream of (X,Y) coordinates for each frame.

Modeled Processes

The model used is directly derived from the model used in the example described in Section 2.5. This model is a degenerate case of the Kalman Filter, mostly because the motion of the fish is generally not very regular. The model is therefore designed such that (near-)linear motion is preferred, however, when linear motion cannot be detected, Euclidean distance to the next observation is used with a score penalty. This means that sitting fish, drifting fish, or fish swimming at a reasonably constant pace could be tracked by comparing the next predicted position with the stream of incoming observations.

Outputs

As output, the system draws a string of colored dots for each observation in a track right on the video frame. This means that a correctly tracked fish leaves a long trail of dots all in the same color. When multiple fish are in the frame, each track gets a different color. Additionally, the model will output a prediction of where each fish is going, showing that tracking continues even when the fish is behind an obstacle.

5.2.3 Results

The annotated video stream looks so intuitive that most casual observers wonder what is so remarkable about it. The application is capable of tracking multiple fish swimming around, crossing paths, and moving in and out of view, while rarely losing track or getting confused. Recorded movies can be obtained at *http://www.pqsnet.net/projects*. For

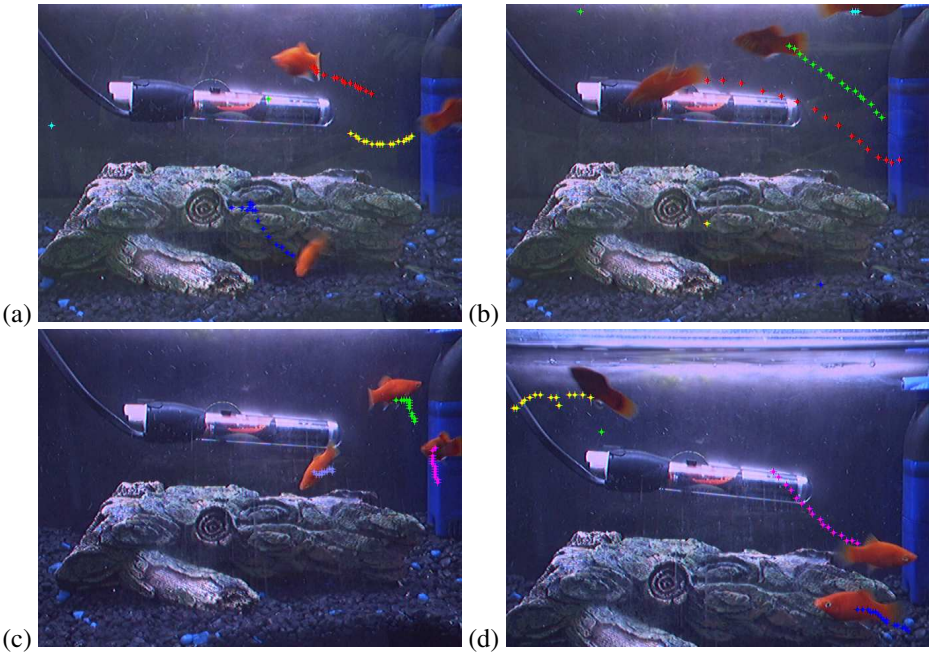


Figure 5.6: Four screen shots of a tank with three fish. Trails are about 16 observations long and show that the past observations have been correlated to belong to the same fish (color of each trail is the same for all past observations).

further details, and an in-depth description of higher level detection of fish behaviors, such as feeding and chasing behavior, refer to Alex's thesis [57]. Several screen shots of the annotated video are shown in Figure 5.6.

5.3 Autonomic Computing

In this section an application of PQS is outlined where the observed environment consists of large server networks. The models describe failure, deteriorating performance, and attack processes. This work was done together with Chris Roblee, who presented a Masters thesis on the subject [103]. A paper was also presented at the second IEEE International Conference on Autonomic Computing in June 2005 [102]. This section aims to give a rough overview of the functionality of the system. For an in-depth description of the specifics, the references should be consulted.

5.3.1 Introduction

Autonomic computing approaches aim to detect deviant behavior of servers and services, and fix the situation before it becomes a problem. Most experimental systems hook into the OS kernel and monitor all system calls for each monitored process [44]. Combined with dynamic behavior learning algorithms, such systems can become an unacceptable load on system resources. Since most failures, degrading performance, and crippling attacks are very process-like in nature it is a logical choice to apply a PQS to this problem domain. For instance, consider a system running a network enabled service to which clients make requests. Examples of such services are web servers, on-line Transaction Processing servers, and Database servers. Assume that this service has a programming flaw that makes the service consume more and more system memory as requests come in over the network. Eventually the system is going to run out of memory and use disk swap, leading to significantly degraded response times. An autonomic monitoring system should have quickly realized that the service process is consuming increasing amounts of memory, and should have taken "appropriate action" before all resources were consumed. Here "appropriate action" would be to stop and restart the buggy service.

5.3.2 Design

The architecture was designed to mimic the conclusions and actions that a diligent administrator would take, given the opportunity to monitor one particular system 24 hours per day.

Sensors

The observations of the system should include most things that an administrator would normally look at. However, another requirement was that the sensors require minimal changes to the monitored host, meaning that hooking into the OS kernel and monitoring system calls was out of the question. Specifically, monitored quantities for the host in general include: total system memory utilization, total CPU usage, process count, and total network usage, and for any directly monitored process includes: process state, process memory utilization, process CPU usage, and process forking behavior. In addition to this host based sensor, we also included information coming from any network IDS sensors.

Modeled Processes

The goal was to look for processes that an administrator would normally look for. This includes unusually high memory or CPU usage, slow response times, and unexpected forking behavior in combination with possible ongoing network attacks, or increased numbers of client requests. For instance, consider monitoring an FTP server. This server may be vulnerable to an unknown exploit that gives the attacker root privileges. Were such an attack to occur the network IDS may or may not pick up the attack signature, however, the host based autonomic computing sensor will report that the FTP server has just forked a shell (such as "/bin/sh"), which would be a really bad sign. Con-

versely, when we consider an SSH server, it would be normal to expect it to spawn shells in response to remote user logins. Although, once again, if a network attack was observed, and for example the CPU utilization of the SSH server was to skyrocket, it would be a sign of a possible compromise of the SSH server.

Outputs

A system cannot be considered autonomic if the detection of failure or attack processes is not acted upon autonomously. The actions that the system invokes should be the same as those that an administrator would normally take in the same situation. A buggy or faulty service may be shut down and restarted safely, thus freeing the resources that it was erroneously holding. This cycle can be repeated forever, as long as the service is restarted before the host system runs out of resources and crashes. In the case of a compromise, however, it is important to shut down the vulnerable service and all its children as soon as possible, without restarting them. An administrator, in such a case, would probably patch the vulnerable service, or replace it by a version that does not have the vulnerability. This, however, lies beyond the capabilities of the PQS based autonomic system. Instead, after disabling the vulnerable service and its children, a human administrator is notified.

5.3.3 Results

The essence of the PQS autonomic server monitoring system was to detect and act as an administrator, but only in the simplest of cases. This required the PQS models to predict the moment of service failure, and detect a possible system compromise. To test the system a service was created that could be configured to be buggy in several different ways: it could leak memory, increase CPU usage with every request, or reduce response times. Additionally, an FTPD service was used that could be exploited to give a root shell, and a web service was used that could be crippled by a DOS attack.

For all three different deteriorating conditions of the buggy service, the system only needed observations from the host based sensor to correctly determine the moment to restart the service. The web server was shut down based on a network IDS observation that an attack had occurred, and the failure of the service to respond to subsequent queries. In retrospect, the web service could safely have been restarted. The FTPD service was correctly shut down after the host based sensor detected that it had spawned a root shell (which was also killed). It must be said, however, that simply killing the service and the shell is only one step in cleaning up after the attack. There is always a slight delay between the compromise, the detection, and the reaction of the system. During this delay an automated attack could have installed a root-kit, thus leaving access to the compromised system open, even though the vulnerable service has been killed. Additional details can be found in the references [103], and [102].

5.4 Application overview

This section gives an overview of many of the applications of PQS. Most of these applications have been discussed in one way or another in this thesis, and all of them use exactly the same underlying PQS software. The diversity of the domains that PQS has been used for is a strong argument for the power of the PQS paradigm.

Bouncing balls (Section 2.6)	
Application Area	Kinematic tracking (of billiard balls)
Sensors	Synthetic (X,Y) pairs
Models	Simplified Kalman Filter
Output	Tracks and predicted positions
Performance Measured How?	Accuracy of predictions

DIB:S (Chapter 3)	
Application Area	Internet worms
Sensors	ICMP Destination Unreachable enabled routers
Models	Fast and medium-fast moving worms
Output	Alerts of aggressively scanning hosts
Performance Measured How?	Infection percentage at detection time

Covert Channels (Section 5.1)	
Application Area	Malicious data exfiltration
Sensors	Data flow sensor, packet timing sensor
Models	Information theory based, channel capacity, statistical
Output	Communications that are suspected covert channels
Performance Measured How?	False positive and false negative rates

Fish tracking (Section 5.2)	
Application Area	Kinematic tracking (of fish)
Sensors	Video centroids
Models	Simplified Kalman Filter
Output	Correlated Path and predicted position
Performance Measured How?	Accuracy of predictions

Autonomic Computing (Section 5.3)	
Application Area	Server monitoring
Sensors	Host based (process and OS), Snort
Models	Service failure, deteriorating performance, service compromise or DOS
Output	Autonomic actions: shut down and restart, terminate
Performance Measured How?	Total number of requests serviced per time period

Appendix A

PQML Specification

A.1 Introduction

The Process Query Modeling Language (PQML) is an assembly inspired, low level language specification for defining observations and models for Process Query Systems (PQS). Any implementation of a Process Query System must be able to parse PQML (pronounce Puh-Que-Mol) files and obtain observation layouts and model definitions from such files.

PQS models work on tracks of observations that are constructed by the core system. These tracks of observations are scored by PQS models through likelihoods. Models are called to evaluate tracks and assign a likelihood score between 0 and 1. These scores are then used by the PQS engine to determine which tracks should be pruned and which tracks get to survive. As more observations arrive the tracks grow and the models are called again to evaluate the track. Models can also be called to evaluate tracks even if that track did not change. This accounts for time passing, so the model can make the track age out. The goal, therefore, of PQML is to provide a structured interface for evaluating observations in a track, and to score that track with a likelihood. This likelihood can be configured to decay logarithmically by setting a halflife value in seconds. This decay will be continuous and when a new likelihood is set it will start decaying immediately.

There is no explicit information given on the binary representation of a PQML definition, it is up to the implementer of the specific PQS to determine what the best internal representation for its PQS is. This specification document lists the requirements for the interpreter and the syntax of the language. The section at the end gives several examples which should be used to verify the correct parsing and execution of the PQML interpreter in any PQS implementation. It is expected that all higher level models can be compiled into a PQML program and so be used by a PQS. Hidden Markov Models, or state machines can have their transition predicates be PQML programs, thus making the translation to PQML more convenient.

Starting November 10^{th}, 2004, PQML includes Arrays in the *.data*, *.observation* and *.conclusion* sections. These arrays must be addressed by special instructions which take an extra argument to index the array. Arrays are now considered part of the base specification.

A.2 Interpreter Specification

This chapter defines the minimum specifications for a PQML interpreter. A PQML interpreter is a virtual machine with registers, a program counter, flags and a stack. There are three sets of registers, one for each datatype:

DATATYPES	
type:	*minimum specification:*
int	signed integer of at least 32 bits
float	at least IEEE 754 floating point number compatible
string	character string of at least 32 characters

A.2.1 Registers

The PQS implementation is required to have at least 32 registers of each datatype: int, float, and string. All registers are general purpose, and instruction opcodes are register specific. Addressing of registers is done starting from 0:

i0—i31	integer registers
f0—f31	floating point registers
s0—s31	string registers

A.2.2 Stack

The stack is used to record the return address in a CALL/RETURN sequence. Additionally, the stack can be explicitly modified with any PUSH/POP instruction pair. The size of the stack may depend on the implementation of the particular PQS, however, it should at least hold 64 items, regardless of the item size. (This is especially relevant regarding the string datatype, since its size is large compared to other dataypes.)

A.2.3 Compare register

The compare register holds the result from the CMP operations. The datatype of the compare register is integer. Conditional jumps query the compare register to adjust program control flow.

A.3 Specification of program sections

PQML consists of six different program sections that can be placed at any location in a PQML file. The *.text* sections contain opcodes and program code, the *.data* sections contain specification of memory, both constants, and track-independent model memory. The special section *.observation* defines the layout and default values of a particular observation type. It is up to the particular PQS implementation to aquire the observations and map them to the format specified in the PQML file. Very similar to observations is the *.conclusion* section. The definition of a conclusion is a stateful section that is kept with each track. When tracks are split off in children this conclusion section is cloned as well. The PQML model can write (and read) these conclusion variables, and track specific data can be stored by the model, to later be used as stateful data. This conclusion section also defines the output of the PQS. The $.halflife$ directive is not really a section because it only takes a floating point value indicating the halflife value in seconds to use to decay the track likelihood. Likelihood decay can be disabled by setting a negative value. Finally, an *.include* section is a one-line directive which can be used to include another PQML file.

Text is parsed in the order that it is received. An include puts the included file right at the point of include. Because there are no explicit line separators, there may only be one data definition, or instruction per line. All data and observations should be parsed first to

identify their labels. Text should be parsed in two passes, the first pass to identify all the labels and verify the syntax, the second pass to generate code. Execution of code starts at the *start* label in the *.text* section. There must be exactly one *start* label in a PQML program.

Comments are prepended by a semi-colon: ";" and run to the end of the line.

A.3.1 Labels

Labels are used to identify data, observation variables, and locations in program code. Labels must start with a letter, but can contain numbers in all but the first character. Labels are case sensitive and may contain up to 32 characters. Labels in a text section are immediately followed by a colon ":".

A.3.2 Data (.data)

Since PQML does not feature immediates in the instruction code, all constant data must be defined in a separate .data section. The labels can then be used in the program code to load these constants into registers, where it can be modified. Storage of data can also be done to this data section, but it is, in general not recommended. Although it possible to write to all memory locations it is advised to keep a very clear distinction between constants and modifiable data. This data section can be used by models that are self-learning and cannot always store model-data with tracks, because the learning data is track independent. Any PQS implementation must be able to hold at least 1000 data definitions per PQML program.

Data is defined by a **datatype label value** triplet, with one data definition per line. Three datatypes are valid (int, float, and string) as defined in the Interpreter Specification section. Strings in a data section must have double quotation marks around it: " and ". Example:

```
.data
        int     cnt0    0
        float   PI      3.1415
        string  str1    "test"
```

A.3.3 Observations (.observation)

Observation specifications define how the *lobs* instructions will be able to access data in observations. All observation types and layouts must be defined in the PQML program before it can be used by the program code. The *.observation* directive takes the observation name directly after the directive. The data layout definition follows on the lines below, each definition followed by a default value. Every *.observation* section defines exactly one observation. The layout definition of the observation follows the syntax defined above for *.data* sections. Any implementation of a PQS must be able to handle at

least 4 different observation definitions per PQML model. Example:

```
.observation packet
                string  src_host    "sky-walker"
                string  dst_host    "artemis"
                int     protocol    6
                int     src_port    35024
                int     dst_port    22
```

A.3.4 Track state and conclusion (.conclusion)

Any PQML program can have only one *.conclusion* section. It defines the layout of the stateful data that is associated with each track. Also, it defines the conclusion that will be published for any given track by the PQS.

When a track of related events gets created by a PQS it must create a stateful section for each PQML model. When a track is cloned and creates offspring, the stateful section must be cloned as well. A PQML model may store stateful information in this section that can be reused on subsequent evaluations of the track. If, for example, the model defines a state machine, then the *.conclusion* stateful section of a track can hold the state that the machine is currently in. This eliminates the need for the PQML model to re-evaluate the entire track. The stored state is enough to determine how the newly added observation is going to affect the current state. This stateful section must be unique for every track of related events in the PQS.

Besides being a stateful, readable and writable section of memory associated with each track, it is also, by definition, the conclusion for that track. This means that when the PQS decides to publish a track as a result the *.conclusion* section associated with that track must be published as well. The PQML model may therefore put current state data, as well as final conclusion (and possibly even predicted state) data in this section. This *.conclusion* track data must be published by the PQS as an observation. This means that the PQS can publish its conclusions directly into a second PQS where the *.conclusion* is defined as a *.observation* input.

The stateful track data can be read and written with the GETC and SETC instructions. The *.conclusion* section defines datatypes, labels, and defaults. Example:

```
.conclusion modelstate
                string  modeltext       "State of Model"
                int     DFA_state       0
                int     predicted_next1 5
                int     predicted_next2 4
                float   next1_chance    0.4
                float   next2_chance    0.1
```

In addition to defining variables that can be assigned in the PQML model there are several variables that may be added to a conclusion section only, and which get filled in

by the tracker on publication. This means that the values are not accessible inside the PQML program.[1] The identifiers are: *intHYPOTHESIS_ID*, *intHYPOTHESIS_SIZE*, and *floatHYPOTHESIS_SCORE*. Type specifications are mandatory and default values must be given, although they are ignored by the publishing logic of the PQS. The hypothesis ID is unique for every hypothesis and changes even when a new observation arrives at a hypothesis. This number can be easily used to regroup conclusion observations into a full hypothesis. Size indicates the number of tracks in the hypothesis, and score returns the averaged score of the hypothesis. Example:

.conclusion modelstate2			
	int	HYPOTHESIS_ID	-1
	int	HYPOTHESIS_SIZE	-1
	float	HYPOTHESIS_SCORE	-1.0

A.3.5 Opcodes and Syntax (.text)

Code sections are identified by the *.text* directive. Labels in the program code are followed directly by a colon ":". Execution starts at the *start* label. Instructions are all lower case. The examples are designed to give good insight into the use of these instructions and building PQML programs.

General			
nop		No operation	nop
exit		Terminate program	exit
Arithmetic			
addi	i_1, i_2	Add integer	$i_1 \leftarrow i_1 + i_2$
addf	f_1, f_2	Add float	$f_1 \leftarrow f_1 + f_2$
subi	i_1, i_2	Subtract integer	$i_1 \leftarrow i_1 - i_2$
subf	f_1, f_2	Subtract float	$f_1 \leftarrow f_1 - f_2$
muli	i_1, i_2	Multiply integer	$i_1 \leftarrow i_1 \times i_2$
mulf	f_1, f_2	Multiply float	$f_1 \leftarrow f_1 \times f_2$
divi	i_1, i_2	Divide integer	$i_1 \leftarrow i_1 / i_2$
divf	f_1, f_2	Divide float	$f_1 \leftarrow f_1 / f_2$
modi	i_1, i_2	Modulo (remainder)	$i_1 \leftarrow i_1 \% i_2$
xltif	i_1, f_2	Typecast F to I	$i_1 \leftarrow$(int) f_2
xltfi	f_1, i_2	Typecast I to F	$f_1 \leftarrow$ (float) i_2

[1] To make these values accessible during model execution would put a serious cap on efficient implementations of a Process Query System. In all cases the different hypotheses that get created are very similar, meaning that they contain many of *the same* tracks. Efficient PQS implementations would not clone these tracks, thus only needing one model evaluation per copy. Such a track would therefore belong to *many* hypotheses at once.

Additional Arithmetic			
sqrtf	f_1	Square root	$f_1 \leftarrow \text{sqrt}(f_1)$
sinf	f_1	Radial Sine	$f_1 \leftarrow \sin(f_1)$
cosf	f_1	Radial Cosine	$f_1 \leftarrow \cos(f_1)$
tanf	f_1	Radial Tangent	$f_1 \leftarrow \tan(f_1)$
asinf	f_1	Radial Arc Sine	$f_1 \leftarrow \text{asin}(f_1)$
acosf	f_1	Radial Arc Cosine	$f_1 \leftarrow \text{acos}(f_1)$
atanf	f_1	Radial Arc Tangent	$f_1 \leftarrow \text{atan}(f_1)$
logf	f_1	Natural Log	$f_1 \leftarrow \ln(f_1)$
expf	f_1	Natural Exp	$f_1 \leftarrow e^{f_1}$
powf	f_1, f_2	Power	$f_1 \leftarrow f_1^{f_2}$
Extended Data			
xchi	i_1, i_2	Exchange integer	$i_1 \leftrightarrow i_2$
xchf	f_1, f_2	Exchange float	$f_1 \leftrightarrow f_2$
xchs	s_1, s_2	Exchange string	$s_1 \leftrightarrow s_2$
movi	i_1, i_2	Copy integer	$i_1 \leftarrow i_2$
movf	f_1, f_2	Copy float	$f_1 \leftarrow f_2$
movs	s_1, s_2	Copy string	$s_1 \leftarrow s_2$
Control flow			
jmp	*label*	Jump to instruction at *label*	PC $\leftarrow$ *label*
be	*label*	Jump if compare was equal	if CMP $= 0$ PC $\leftarrow$ *label*
bne	*label*	Jump if not equal	if CMP $\neq 0$ PC $\leftarrow$ *label*
bg	*label*	Jump if greater	if CMP > 0 PC $\leftarrow$ *label*
bge	*label*	Jump if greater or equal	if CMP ≥ 0 PC $\leftarrow$ *label*
bl	*label*	Jump if lesser	if CMP < 0 PC $\leftarrow$ *label*
ble	*label*	Jump if lesser or equal	if CMP ≤ 0 PC $\leftarrow$ *label*
call	*label*	Push PC on stack and jump	push(PC), PC $\leftarrow$ *label*
ret		Pop PC from stack	PC $\leftarrow$ pop()
Compares			
cmpi	i_1, i_2	Compare integer	CMP $\leftarrow i_1 - i_2$
cmpf	f_1, f_2	Compare float	CMP $\leftarrow$ (int) $f_1 - f_2$
cmps	s_1, s_2	Compare string	CMP $\leftarrow$ strcmp(s_1, s_2)
Data section operations			
geti	i_1, *label*	Get integer	$i_1 \leftarrow$ DATA[*label*]
getf	f_1, *label*	Get float	$f_1 \leftarrow$ DATA[*label*]
gets	s_1, *label*	Get string	$s_1 \leftarrow$ DATA[*label*]
seti	i_1, *label*	Set integer	DATA[*label*] $\leftarrow i_1$
setf	f_1, *label*	Set float	DATA[*label*] $\leftarrow f_1$
sets	s_1, *label*	Set string	DATA[*label*] $\leftarrow s_1$

Stack			
pushi	i_1	Push integer	push(i_1)
pushf	f_1	Push float	push(f_1)
pushs	s_1	Push string	push(s_1)
popi	i_1	Pop integer	$i_1 \leftarrow$ pop()
popf	f_1	Pop float	$f_1 \leftarrow$ pop()
pops	s_1	Pop string	$s_1 \leftarrow$ pop()

The observation instructions below operate directly on the observations in a track. The second parameter to these instructions is formatted with a *type*, an integer register, and a *label* field. The type is as identified by *.observation* sections, as are the label fields. The type refers to a specific type of observation, and the label refers to the particular named field inside the observation. The integer register indicates which observation the data is going to be retrieved from, where 0 is the last observation that came in, of the specified type. The first observation of a particular type in a track can be retrieved with the *ldsize* instruction outlined further below. The *lobsts* instruction is a special case where the timestamp of the requested observation is returned in two destination registers. The first will hold the number of seconds since the Epoch (00:00:00, January 1^st, 1970), the second register will hold the number of microseconds in this second. As a type specifier *ANY* may be used to get the timestamp from the last observation added to this track. For comparison the *lnowts* loads the current time (now) into the two destination registers.

Observations	
lobsi	i_1, *type*[i_2].*label*
lobsf	f_1, *type*[i_2].*label*
lobss	s_1, *type*[i_2].*label*
lobsts	i_1, i_2, *type*[i_3]
lnowts	i_1, i_2

The instructions below set or get the likelihood for the track. When the program is called by the PQS the likelihood that is set (and thus returned on a get) should be the likelihood that was previously given to this track by this particular model minus the logarithmic decay set by the halflife parameter.

Likelihood modification			
getl	f_1	Get likelihood	$f_1 \leftarrow$ LIKELIHOOD
setl	f_1	Set likelihood	LIKELIHOOD $\leftarrow f_1$

The conclusion instructions load and store stateful data with the track under evaluation. The data stored in a conclusion section remains with the track (and all its children, which get their own copy) and can be retrieved and updated on subsequent calls to the PQML model. This section is also published as the conclusion for the given track. For more information refer to the section regarding the *.conclusion* directive.

Track state and conclusion.		
getci	i_1, *concl_label*	$i_1 \leftarrow$ T.concl(*label*)
getcf	f_1, *concl_label*	$f_1 \leftarrow$ T.concl(*label*)
getcs	s_1, *concl_label*	$s_1 \leftarrow$ T.concl(*label*)
setci	i_1, *concl_label*	T.concl(*label*) $\leftarrow i_1$
setcf	f_1, *concl_label*	T.concl(*label*) $\leftarrow f_1$
setcs	s_1, *concl_label*	T.concl(*label*) $\leftarrow s_1$

The instructions below are special function instructions. The *ldsize* instruction loads the number of *type* observations in the current track into the given integer register. To determine the size of the entire track (all observations) the type specifier should be set to *ALL*. The *cmpt* instruction compares the type of the i_1th observation in the track to the given *type*. If the types match the compare register is set to 1, else the compare register is set to 0. The integer register holds the number of the observation to compare to, where 0 is the last observation that was added to this track. The *tmod* instruction sets the compare register to 1 if the track was modified since the last time this model was called to evaluate this track. If no new observation was added to this track since the last evaluation, then the compare register is set to 0.

Special instructions			
ldsize	i_1, *type*	return number of *type* obsv.	$i_1 \leftarrow$ #*type*
cmpt	i_1, *type*	compare i_1th obsv. to *type*	CMP $\leftarrow$ obs[i_1] - *type*
tmod		set if track was modified	if mod(TRACK) CMP $\leftarrow$ 1

A.3.6 Logarithmic Likelihood decay (.halflife)

The *.halflife* directive sets that speed with which track likelihoods will decay over time. After the specified time the likelihood of the track will have decayed to half of its original value, if it wasn't overwritten by a new *setl* instruction. To disable the likelihood decay set a negative value. The *.halflife* directive takes a floating point argument indicating the halflife time in seconds.

A.3.7 Including other files (.include)

The *.include* directive takes one argument which must be a valid filename of another PQML file. This file can contain data, observation, and text sections, as well as other inclusions. It is the task of the programmer to ensure that the includes do not redefine labels (data, text, or observation) or contain cyclic includes.

A.3.8 Arrays

As of November 10^{th}, 2004, PQML includes Arrays in the *.data*, *.observation* and *.conclusion* sections. This section describes how to specify arrays, and how to address

them through any of the 15 new special instructions. Arrays are specified by a size parameter placed between square brackets *directly* following (no whitespace) the name label. The size must be specified and cannot be variable. The value specified after the array name label will be given to all elements in the array as the default value. Only the *int* and $float$ formats are valid for arrays.

```
.data
                        int     state_space[10]   -1

.conclusion HMMstate
                        float   state_space[4]    0.25
```

To reference elements of an array an integer register must be given that holds the index. The indexing is similar to the C programming language: 0 addresses the first element, 1 addresses the second, and so forth.

Data section array operations		
getia	i_1, *label*[i_2]	$i_1 \leftarrow$ DATA[*label*@i_2]
getfa	f_1, *label*[i_2]	$f_1 \leftarrow$ DATA[*label*@i_2]
setia	i_1, *label*[i_2]	DATA[*label*@i_2] $\leftarrow i_1$
setfa	f_1, *label*[i_2]	DATA[*label*@i_2] $\leftarrow f_1$
Track state and conclusion array oprators.		
getcia	i_1, *concl_label*[i_2]	$i_1 \leftarrow$ T.concl(*label*@i_2)
getcfa	f_1, *concl_label*[i_2]	$f_1 \leftarrow$ T.concl(*label*@i_2)
setcia	i_1, *concl_label*[i_2]	T.concl(*label*@i_2) $\leftarrow i_1$
setcfa	f_1, *concl_label*[i_2]	T.concl(*label*@i_2) $\leftarrow f_1$
Observations array operators		
lobsia	i_1, *type*[i_2].*label*[i_3]	
lobsfa	f_1, *type*[i_2].*label*[i_3]	

Appendix B

PQML Models

B.1 funcs.pqml

This code is sourced by several of the PQML code files given in this appendix. It implements a function that translates two integer time pairs (seconds and microseconds) to a floating point time difference in seconds.

```
;
;  Vincent Berk
;  [10/28/2004]
;
;  Take notice which registers are used for parameters,
;  and which are used for return values.  Make sure to
;  push registers which you don't want overwritten accidentally.
;

    ;  Some constants used here

.data
    float   funcs_million   999999

    ;
    ;  This function calculates the delta T in seconds
    ;  from four integer registers, and returns the
    ;  result in one float register.
    ;     Parameters:
    ;           i20     -  seconds old
    ;           i21     -  microseconds old
    ;           i22     -  seconds new
    ;           i23     -  microseconds new
    ;     Returns:
    ;           f20     -  difference in seconds
    ;

.text
delta_t:
    pushf   f21             ; save f21
    pushi   i20             ; save i20
    pushi   i21             ; save i21
    pushi   i22             ; save i22
    pushi   i23             ; save i23

    subi    i22, i20        ; secs new - secs old
    subi    i23, i21        ; usecs new - usecs old
    xltfi   f20, i23        ; usecs to float

    getf    f21, funcs_million
    divf    f20, f21        ; usecs to secs

    xltfi   f21, i22        ; seconds
    addf    f20, f21        ; add the seconds difference

    popi    i23             ; restore i23
    popi    i22             ; restore i22
    popi    i21             ; restore i21
    popi    i20             ; restore i20
```

```
    popf    f21             ; restore f21
    ret                     ; done
```

B.2 movingdots.pqml

This PQML model program was used for the example in Section 2.5. The model scores tracks based on how well the predicted current position of the dot matches the received observation.

```
;
;  PQML for tracking a moving dot
;  If the length of the momentum vector is larger than 200,
;  the track gets a score of 0.  No exceptions.
;
;  Vincent Berk
;  [10/28/2004]
;
;  It is expected that tracks < 2 are not pruned, just
;  to give them a chance to grow.
;
    ;
    ;  After 5 seconds of lack of new observations
    ;  the track is half as accurate/valid.
    ;

.halflife 5.0

    ;
    ;  Get some useful functions
    ;

.include funcs.pqml

    ;
    ;  An observation contains just an X and Y
    ;  position.
    ;

.observation position
    int     pos_x       0       ; ranges from 0 to 1000
    int     pos_y       0       ; idem

    ;
    ;  Track dynamic data.  When the object_number is still
    ;  zero no track number was yet assigned to this track.
    ;  It is like the color of a tracked object.
    ;

.conclusion direction
    int     cur_pos_x       0
    int     cur_pos_y       0
    float   momentum_x      1.0
```

```
    float   momentum_y      1.0
    int     object_number    -1

    ;
    ;  Dynamic data which is track specific.  Every time
    ;  a track is started that does not have an object_number
    ;  this number is incremented and assigned to that track.
    ;

.data
    int     current_cnt     0

        ;
        ;  Consts, static data:
        ;

    int     c0      0
    int     c1      1
    int     c2      2
    float   f_zero          0.0
    float   f_one           1.0
    float   f_ten           10.0
    float   f_half           0.5
    float   f_thres        200.0

;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;  The actual toplevel model
;

.text
start:
        ;
        ;  Load constants
        ;

    geti    i0, c0
    geti    i1, c1

        ;
        ;  Two possibilities:  new observation or no new observation
        ;

    tmod
    be      no_new

    call    new_observation
    jmp     end

no_new:
    call   nonew_observation
    jmp    end

;;;;;;;;;;;
;
```

```
;  The end.
;

end:
    exit

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;  Function to calculate the new score
;  and the new momentum.
;  Check the length of this track first.
;

new_observation:

    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ;
    ;  First estimate the current position based on
    ;  the existing momentum
    ;
        ;
        ;  Constants
        ;

    geti    i0, c0
    geti    i1, c1

        ;
        ;  Size
        ;

    ldsize      i2, position
    setci       i2, object_number
    cmpi        i2, i1          ; size - 1
    ble         set_zero        ; cmp <= 0, set momentum to zero.

        ;
        ;  Calculate time difference
        ;

    lobsts  i20, i21, position[i1]
    lobsts  i22, i23, position[i0]

    call    delta_t         ;  f20 < delta_t

        ;
        ;  Load previous last position
        ;

    lobsi   i3, position[i1].pos_x     ; previous x
    lobsi   i4, position[i1].pos_y     ; previous y

    xltfi   f3, i3                  ; f3 <- i3
    xltfi   f4, i4                  ; f4 <- i4
```

```
movf    f7, f3      ;  backup previous position x
movf    f8, f4      ;  backup previous position y

    ;
    ;  Multiply momentum by delta t and add to original
    ;  position
    ;

getcf   f5, momentum_x
getcf   f6, momentum_y
mulf    f5, f20
mulf    f6, f20
addf    f7, f5
addf    f8, f6

    ;
    ;  Estimated new position is now in f3,f4
    ;  Get the new observation, and calculate the
    ;  difference between the estimated and the
    ;  actual position.
    ;

lobsi   i5, position[i0].pos_x     ; newest x
lobsi   i6, position[i0].pos_y     ; newest y

xltfi   f5, i5                  ; f5 <- i5
xltfi   f6, i6                  ; f6 <- i6

    ;
    ;  Calculate the difference between estimated and
    ;  actual position.  Euclidian distance.
    ;

subf    f7, f5
subf    f8, f6
mulf    f7, f7
mulf    f8, f8
addf    f7, f8
sqrtf   f7          ;  This is the Euclidian distance.

    ;
    ;  Make a score out of this absolute distance
    ;  by multiplying by 10 -- this is a normalisation!
    ;

getf    f8, f_ten
divf    f7, f8
divf    f7, f8
divf    f7, f8
getf    f8, f_one
subf    f8, f7     ;  f8 now contains the new score

    ;
    ;  If track-len == 2, set score to 0.5
    ;
```

```
    push    i0
    geti    i0, c2
    cmpi    i2, i0
    bne     set_normal

set_half:
    getf    f8, f_half
    jmp     set_normal

set_normal:
    setl    f8
    pop     i0

    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ;
    ;  Now it is time to update the position to reflect
    ;  the actual new observation.  Also update the
    ;  momentum.  Register values:
    ;   (f3,f4)  previous position
    ;   (f5,f6)  new(current) position float
    ;   (i5,i6)  new(current) position int
    ;   f20      time difference between old and new
    ;
        ;
        ;  Momentum.
        ;

    subf    f5, f3
    subf    f6, f4
    divf    f5, f20
    divf    f6, f20
    setcf   f5, momentum_x
    setcf   f6, momentum_y

        ;
        ;  Check the length of the momentum vector.
        ;  If it is larger than 200, then set score to zero.
        ;

    getcf   f11, momentum_x
    getcf   f12, momentum_y
    mulf    f11, f11
    mulf    f12, f12
    addf    f11, f12
    sqrtf   f11
    getf    f12, f_thres
    cmpf    f11, f12
    bge     set_zero_score
    jmp     cont

set_zero_score:
    getf    f8, f_zero
    setl    f8
```

```
cont:
        ;
        ;  Set new position.
        ;

    setci   i5, cur_pos_x
    setci   i6, cur_pos_y

        ;
        ;  Done.
        ;

    jmp     momentum_end

        ;
        ;  Set momentum to zero
        ;

set_zero:

    getf    f0, f_zero
    setcf   f0, momentum_x
    setcf   f0, momentum_y

        ;
        ;   Return.
        ;

momentum_end:

    ret

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;  Update the current estimated position
;  when no new observation was received.
;

nonew_observation:

        ;
        ;  Constants
        ;

    geti    i0, c0
    geti    i1, c1

        ;
        ;  Calculate time difference
        ;

    lobsts  i20, i21, position[i0]
    lnowts  i22, i23

    call    delta_t         ;  f20 < delta_t
```

```
            ;
            ;  Load last position
            ;

        lobsi   i3, position[i0].pos_x     ; newest x
        lobsi   i4, position[i0].pos_y     ; newest y

        xltfi   f3, i3                  ; f3 <- i3
        xltfi   f4, i4                  ; f4 <- i4

            ;
            ;  Multiply momentum by delta t and add to original
            ;  position
            ;

        getcf   f5, momentum_x
        getcf   f6, momentum_y
        mulf    f5, f20
        mulf    f6, f20
        addf    f3, f5
        addf    f4, f6

            ;
            ;  Translate the new floating point position back
            ;  to integers
            ;

        xltif   i3, f3
        xltif   i4, f4
        setci   i3, cur_pos_x
        setci   i4, cur_pos_y

            ;
            ;  Done with this.
            ;

        ret

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;  End of model
;
```

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[2] Kamran Ahsan. Covert channel analysis and data hiding in TCP/IP. *Master's thesis, University of Toronto*, 2000.

[3] Kamran Ahsan and Deepa Kundur. Practical data hiding in TCP/IP. *Multimedia and Security Workshop at ACM Multimedia 2002*, 2002.

[4] Symantec security response - BackOrifice Trojan, February 18, 2003. Available at http://securityresponse.symantec.com/avcenter/venc/data/backorifice.trojan.html.

[5] F. Baker. RFC 1812: Requirements for IP version 4 routers. volume 1812 of *Request for Comments*. 1995.

[6] George Bakos and Vincent Berk. Early detection of internet worm activity by metering ICMP destination unreachable messages. In *Proceedings of the SPIE Aerosense*, 2002.

[7] J.S. Baras, A.A. Cardenas, and V. Ramezani. On-line detection of distributed attacks from space-time network flow patterns. In *23rd Army Science Conference*, December 2002.

[8] Vincent Berk, George Bakos, and Robert Morris. Using sensor networks and data fusion for early detection of active worms. In *Proceedings of the IEEE International Workshop on Information Assurance (IWIA 2003)*, Darmstadt Germany, March 2003.

[9] Vincent Berk, Wayne Chung, Valentino Crespi, George Cybenko, Robert Gray, Diego Hernando, Guofei Jiang, Han Li, and Yong Sheng. Process query systems for surveillance and awareness. In *Proceedings of the 7th World Multifconference on Systems, Cybernetics and Informatics (SCI 2003)*, Orlando, Florida, July 2003.

[10] Vincent Berk and Naomi Fox. Process query systems for network security monitoring. In *Proceedings of the SPIE Vol. 5778, Sensors, and Command, Control, Communications, and Intelligence (C3I)*, April 2005.

[11] Vincent H. Berk, George Cybenko, and Robert S. Gray. Early detection of active internet worms. In Vipin Kumar, Jaideep Srivastava, and Aleksander Lazarevic, editors, *Managing Cyber Threats*, chapter 6, pages 147–180. Springer, 2005.

[12] Vincent H. Berk, Robert S. Gray, and George Bakos. Using sensor networks and data fusion for early detection of active worms. In *Proceedings of AeroSense 2003: SPIE's 17th Annual International Symposium on Aerospace/Defense Sensing, Simulation, and Controls*, volume 5071, Orlando, Florida, April 2003.

[13] Richard E. Blahut. Computation of channel capacity and rate-distortion functions. *IEEE Transactions on Information Theory*, IT-18(4):460–473, July 1972.

[14] Aniruddha Bohra, Iulian Neamtiu, Pascal Gallard, Florin Sultan, and Liviu Iftode. Remote repair of operating system state using Backdoors. In *Proceedings of the International Conference on Autonomic Computing*, May 2004.

[15] Robert Brown. A brief account of microscopical observations made in the months of June, July and August, 1827, on the particles contained in the pollen of plants; and on the general existence of active molecules in organic and inorganic bodies. *Edinburgh New Philosophical Journal*, pages 358–371, 1828.

[16] Robert G. Brown and Patrick Y.C. Hwang. *Introduction to Random Signals and Applied Kalman Filtering*. John Wiley & Sons, 1983.

[17] Daniel J. Burroughs, Linda F. Wilson, and George V. Cybenko. Analysis of distributed intrusion detection systems using Bayesian methods. In *Proceedings of the 21st IEEE International Performance, Computing and Communications Conference (IPCCC 2002)*, April 2002.

[18] David R. Butenhof. *Programming with POSIX Threads*. Addison Wesley, May 2003.

[19] Joao B. D. Cabrera, Lundy Lewis, Xinzhou Qin, Wenke Lee, and Raman K. Mehra. Proactive intrusion detection and distributed denial of service attacks – a case study in security management. *Journal of Network and Systems Management*, 10(2), June 2002.

[20] Serdar Cabuk, Carla Brodley, and Clay Shields. IP Covert Timing Channels: Design and Detection. *Proceedings of the 11th ACM conference on Computer and Communications Security*, 2004.

[21] Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic, 1999. ISBN 0-7923-8609-4.

[22] David M. Chess and Steve R. White. An undetectable computer virus. *Virus Bulletin Conference*, September 2000.

[23] Steven Cheung, Rick Crawford, Mark Dilger, Jeremy Frank, Jim Hoagland, Karl Levitt, Jeff Rowe, Stuart Staniford-Chen, Raymond Yip, and Dan Zerkle. The design of GrIDS: A graph-based intrusion detection system. Technical Report CSE-99-2, Department of Computer Science at UC Davis, 1999.

[24] Douglas Comer. *Internetworking with TCP/IP Principles, Protocols, and Architectures*, volume 1. Prentice Hall, fourth edition, 2000.

[25] Common Vulnerabilities and Exposures List. CVE advisory CAN-2004-0942 apache webserver denial of service vulnerability. Technical report, The MITRE Corporation, 2004.

[26] S.A. Cook. The complexity of theorem proving procedures. In *Proceedings of the third Annual ACM Symposium on Theory of Computing*, pages 151–156, 1971.

[27] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley Series in Telecommunications. John Wiley & Sons, New York, NY, USA, 1991.

[28] Valentino Crespi, Wayne Chung, and Alex B. Jordan. Decentralized sensing and tracking for uav scheduling. In *Proceedings of the SPIE Vol. 5403, Sensors, and Command, Control, Communications, and Intelligence (C3I)*, April 2004.

[29] J. P. Crutchfield. Information and Its Metric. *Nonlinear Structures in Physical Systems – Pattern Formation, Chaos, and Waves*, 1990. 119-130.

[30] J. P. Crutchfield and D. P. Feldman. Regularities unseen, randomness observed: Levels of entropy convergence. *CHAOS (submitted). Santa Fe Institute Working Paper 01-02-012*, 2001.

[31] George Cybenko, Vincent H. Berk, Valentino Crespi, Robert S. Gray, and Guofei Jiang. An overview of process query systems. In *Proceedings of SPIE Vol. 5403 Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security and Homeland Defense III , Orlando, Florida*, April 2004.

[32] D.J. Daley and J. Gani. *Epidemic Modeling*. Cambridge University Press, 1999.

[33] William James Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.

[34] H. G. Dehling and J. N. Kalma. *Kansrekening het zekere van het onzekere*. Epsilon Uitgaven, Utrecht, 1995.

[35] Dshield - distributed intrusion detection system, the internet's early warning system, 2005. Available at http://www.dshield.org.

[36] Bob DuCharme. *XML: The Annotated Specification*. Prentice Hall, 1999.

[37] Gary Dudley, Neeraj Joshi, David M. Ogle, Balan Subramanian, and Brad Topol. Autonomic self-healing systems in a cross-product IT environment. In *Proceedings of the International Conference on Autonomic Computing*, May 2004.

[38] Microsoft SQL Sapphire worm analysis, 2003. Available at http://www.eeye.com/html/Research/Flash/AL20030125.html.

[39] Mark W. Eichin and Jon A. Rochlis. With microscope and tweezers: An analysis of the Internet virus of November 1988. In *Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy*, May 1989.

[40] Wei Fan, Matt Miller, Sal Stolfo, Wenke Lee, and Phil Chan. Using artificial anomalies to detect known and unknown network intrusions. In *Proceedings of the First International Conference on Data Mining*, November 2001.

[41] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial*. Addison Wesley, February 2003.

[42] Tiago Ferreto, Cesar De Rose, and Luiz De Rose. RVision: An open and high configurable tool for cluster monitoring. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002.

[43] G. D. Forney. The Viterbi algorithm. In *Proceedings of the IEEE*, volume 61(3), pages 268–278, 1973.

[44] Stephanie Forrest, Steven Hoffmeyr, Anil Somayaji, and Thomas Longstaff. A sense of self for unix processes. In *IEEE Symposium on Security and Privacy*, pages 120–128, 1996.

[45] Annarita Giani. Efficiency and accuracy trade-offs in process detection. In *Proceedings of the SPIE Vol. 5403, Sensors, and Command, Control, Communications, and Intelligence (C3I)*, April 2004.

[46] J. Giles and B. Hajek. An Information-theoretic and Game-theoretic Study of Timing Channels. *IEEE Transactions on information Theory*, 2002.

[47] Robert S. Gray and Vincent H. Berk. Rapid detection of worms using ICMP-T3 analysis. In *Proceedings of the SPIE Vol. 5403, Sensors, and Command, Control, Communications, and Intelligence (C3I)*, April 2004.

[48] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman, third edition, 2003.

[49] Diego Hernando and Valentino Crespi. Sampling theory for process detection with applications to surveillance and tracking. In *Proceedings of the SPIE Vol. 5403, Sensors, and Command, Control, Communications, and Intelligence (C3I)*, April 2004.

[50] John Hertz, Anders Kroch, and Richard G. Palmer. *Introduction to the Theory of Neural Computation*. Addison Wesley, 1991.

[51] http://www.caida.org/analysis/security/code red/coderedv2_analysis.xml. The spread of the code-red worm (crv2).

[52] http://www.caida.org/analysis/security/code red/index.xml. Caida analysis of code-red. 2002.

[53] http://www.incidents.org/react/code_red.html. Code red. 2001.

[54] Aapo Hyvrinen. Survey on independent component analysis. In *Neural Computing Serveys*, volume 2, pages 94–128, 1999.

[55] Guofei Jiang. Weak process models for robust process detection. In *Proceedings of the SPIE Vol. 5403, Sensors, and Command, Control, Communications, and Intelligence (C3I)*, April 2004.

[56] Guofei Jiang and George Cybenko. Temporal and spatial distributed event correlation for network security. In *Proceedings of the IEEE American Control Conference*, Boston MA, June 2004.

[57] Alex B. Jordan. Models for tracking and level 2 fusion. Master's thesis, Thayer School of Engineering at Dartmouth College, May 2005.

[58] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[59] R.M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103, 1972.

[60] Robert. A. Kemmerer. Shared Resource Matrix Methodology: An Approach to Identifying Storage and Timing Channels: Twenty years later. *Proceedings of the 18th Annual Computer Security Applications Conference*, 2002.

[61] Robert. A. Kemmerer. Shared Resource Matrix Methodology: An Approach to Identifying Storage and Timing Channels. *ACM Transaction on Computer Systems*, August 1983.

[62] Jeffrey Kephart and Steve White. Directed-graph epidemiological models of computer viruses. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, May 1991.

[63] Jeffrey O. Kephart. A biologically inspired immune system for computers. *Artificial Life IV*, 1994.

[64] Olaf Kirch. *LINUX Network Administrators Guide*. O'Reilly, first edition, March 1995.

[65] Helmut Kopka and Patrick W. Daly. *A Guide To LATEX*. Addison Wesley, third edition, 1999.

[66] O. Patrick Kreidl and Tiffany M. Frazier. Feedback control applied to survivability: A host-based autonomic defense system. *IEEE Transactions on Reliability*, 53(1):148–166, March 2004.

[67] Jamie Lerner. *Advanced System and Security Monitoring - Achieving Complete System Control*. JJ Labs Inc., 2003. White Paper, http://www.jjlabs.com.

[68] Donald Lewine. *POSIX Programmers Guide*. O'Reilly and Associates, Inc., first edition, March 1994.

[69] Michael Liljenstam, David M. Nicol, Vincent H. Berk, and Robert S. Gray. Simulating realistic network worm traffc for worm warning system design and testing. In *ACM Workshop on Rapid Malcode, Washington DC*, Oktober 2003.

[70] Michael Liljenstam, Yougu Yuan, BJ Premore, and David Nicol. A mixed abstraction level simulation model of large-scale Internet worm infestations. In *Proceedings of Tenth IEEE/ACM International Conference on Modeling, Analysis and Simulation of Computer and Communications Systems (MASCOTS 2002)*, October 2002.

[71] G. Mansfield, K. Ohta, Y. Takei, N. Kato, and Y. Nemoto. Towards trapping wily intruders in the large. In *Proceedings of the second international workshop on Recent Advances in Intrusion Detection (RAID 1999)*, 1999.

[72] A. A. Markov. Extension of the limit theorems of probability theory to a sum of variables connected in a chain. 1971. Reprinted in Appendix B of: R. Howard. Dynamic Probabilistic Systems, volume 1: Markov Chains.

[73] Stuart McLure, Joel Scambray, and George Kurtz. *Hacking Exposed*. Osborne/McGraw-Hill, third edition, 2001.

[74] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, third edition.

[75] David Moore, Vern Paxon, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. The spread of the Sapphire/Slammer worm. Technical report, CAIDA, 2003.

[76] David Moore, Colleen Shannon, and Jeffery Brown. Code Red: A case study on the spread and victims of an Internet worm. In *Proceedings of the Second Internet Measurement Workshop (IMW 2002)*, November 2002.

[77] David Moore, Colleen Shannon, Geoffrey M. Voelker, and Stefan Savage. Internet quarantine: Requirements for containing self-propagating code. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003)*, April 2003.

[78] David Moore, Geoffrey Voelker, and Stefan Savage. Inferring internet denial-of-service activity. *Usenix*, 2001.

[79] I. Moskowitz, R. Newman, D. Crepeau, and A. Miller. Covert channels and anonymizing networks. *Proceedings of the Workshop on Privacy in the Electronic Society (WPES 03), Washington, DC, USA.*, October 2003.

[80] I. S. Moskowitz and M. H. Kang. Covert channels here to stay? *Proceedings of COMPASS*, 1994.

[81] I. S. Moskowitz and A. R. Miller. The Channel Capacity of a Certain Noisy Timing Channel. *IEEE Transaction on information Theory, Vol 38, n.4*, 1992.

[82] I. S. Moskowitz and A. R. Miller. Simple timing channels. *Proceedings of IEEE Computer Society Symposium on Research in Security and Privacy, Oakland*, 1994.

[83] T. Narten and R. Draves. RFC 3041: Privacy extensions for stateless address autoconfiguration in IPv6. volume 3041 of *Requests For Comments*. 2001.

[84] The netfilter/IPtables project homepage, 2005. Available at http://www.netfilter.org.

[85] Peter G. Neumann and Phillip A. Porras. Experimence with EMERALD to date. In *Proceedings of the First USENIX Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, California, April 1999.

[86] NimSoft. *NimBUS for Server Monitoring*. Solution Overview Paper, http://www.nimsoft.com.

[87] Glenn Nofsinger and Keston Smith. Plume source detection using a process query system. In *Proceedings of the SPIE Vol. 5403, Sensors, and Command, Control, Communications, and Intelligence (C3I)*, April 2004.

[88] Stephen Northcutt and Judy Novak. *Network Intrusion Detection*. New Riders, second edition, September 2000.

[89] N. Ogurtsov, H. Orman, R. Schroeppel, S. O'Malley, and O. Spatscheck. Covert Channel Elimination Protocols. *Technical Reports TR96-14. Department of Computer Science, University of Arizona*, 1996.

[90] Lathi B. P. *Modern Digital and Analog Communication Systems. Third Edition*. Oxford University Press, 1998.

[91] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.

[92] Steve Parker. A simple equation: IT on = business on. *The IT Journal, Hewlett Packard*, 2001.

[93] Vern Paxson. End-to-end routing behavior in the Internet. *IEEE/ACM Transactions on Networking*, 5(5):601–615, 1997.

[94] Vern Paxson and Sally Floyd. Wide-area traffic: The failure of Poisson modeling. In *Transaction on Networking*, pages 226–244. IEEE/ACM, 1995.

[95] D. C. Plummer. RFC 826: An ethernet address resolution protocol. volume 826 of *Request for Comments*. 1982.

[96] Phillip Porras and Alfonso Valdes. Live traffic analysis of TCP/IP Gateways. In *ISOC Symposium on Network and Distributed Systems Security*, 1998.

[97] J. Postel. RFC 792: Internet Control Message Protocol. volume 792 of *Request for Comments*. 1981.

[98] Xinzhou Qin, David Dagon, Guofei Gu, Wenke Lee, Mike Warfield, and Pete Allor. Worm detection using local networks. Submitted to USENIX 2004 and made available on several security mailing lists, 2004.

[99] Lawrence R. Rabiner. A tutorial on Hidden Markov Models and selected applications in speech recognition. *Proceeding of the IEEE*, 77, Num. 2:257–286, 1989.

[100] D. B. Reid. An algorithm for tracking multiple targets. *IEEE Transactions on Automatic Control*, AC-24:843–854, December 1979.

[101] Y. Rekhter and T. Li. RFC 1771: A border gateway protocol 4 (BGP-4). volume 1771 of *Request for Comments*. 1995.

[102] Christopher Roblee, Vincent Berk, and George Cybenko. Large-scale autonomic server monitoring using process query systems. In *Proceedings of the second IEEE International Conference on Autonomic Computing (ICAC-05)*, June 2005.

[103] Christopher D. Roblee. Process query systems for network self awareness: A principled, scalable approach to rapid autonomic healing. Master's thesis, Thayer School of Engineering at Dartmouth College, May 2005.

[104] Stuart Russel and Peter Norvig. *Artificial Intelligence*. Prentice Hall, 1995.

[105] Kristina Lisa Shalizi, Cosma Rohilla Shalizi, and J. P. Crutchfield. Pattern Discovery in Time Series, part I and II: Implementation, Evaluation, and Comparison. *Journal of Machine Learning Research*, 2002.

[106] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, July 1948. Continued 27(4):623-656, October 1948.

[107] Snort - the open source network intrusion detection system, 2005. Available at http://www.snort.org.

[108] Matthew J. Sottile and Ronald G. Minnich. Supermon: A high-speed cluster monitoring system. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2002.

[109] Eugene Spafford. The Internet worm incident. In *Proceedings of the Second European Software Engineering Conference (ESEC '89)*, volume 87 of *Lecture Notes in Computer Science*, pages 446–468. Springer-Verlag, September 1989.

[110] Eugene H. Spafford. An analysis of the Internet worm. In *Proceedings of the 1989 European Software Engineering Conference*, September 1989.

[111] Eugene H. Spafford. The Internet worm: Crisis and aftermath. *Communications of the ACM*, 32(6), June 1989.

[112] Stuart Staniford. Analysis of the spread of the July infestation of the Code Red worm. technical report, Silicon Defense, 2001.

[113] Stuart Staniford, James A. Hoagland, and Joseph M. McAlerney. Practical automated detection of stealthy portscans. *Journal of Computer Security*, 10:105–136, 2002.

[114] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the Internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium (Security '02)*, San Francisco, California, August 2002.

[115] W. Richard Stevens. *UNIX Network Programming*, volume 1. Prentice Hall PTR, second edition, 1998.

[116] W. Richard Stevens. *UNIX Network Programming*, volume 2. Prentice Hall PTR, second edition, 1998.

[117] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison Wesley, January 2003.

[118] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, September 2001.

[119] Symantec security response - backdoor SubSeven, July 25, 2003. Available at http://www.symantec.com/avcenter/venc/data/backdoor.subseven.html.

[120] Kenneth Theriault, Wilson Farrel, Helene Henri, Derrick Kong, and William Nelson. The SARA experiment: Coordinated autonomic defense against an e-mail-borne virus. In *Proceedings of the 2002 IEEE Workshop on Information Assurance*, June 2002.

[121] Tripwire, inc. - tripwire is the leading provider of change auditing solutions, 2005. Available at http://www.tripwire.com.

[122] K. S. Trivedi. *Probability, Statistics with Reliability, Queueing and Computer Science Applications*. Wiley, second edition, 2002. ISBN 0-471-33341-7.

[123] Norbert Wiener. Extrapolation, interpolation and smoothing of stationary time series with engineering applications. 1949.

[124] Matthew M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. Technical Report 172, HP Labs Bristol, 2002.

[125] Derrick Wood. *Theory of Computation*. John Wiley and sons, 1987.

[126] Derrick Wood. *Data Structures, Algorithms, and Performance*. Addison Wesley, 1993.

[127] Vinod Yagneswaran, Paul Barford, and Johannes Ullrich. Internet intrusions: Global characteristics and prevalance. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS 2003), San Diego California*, June 2003.

[128] Cliff C. Zou, Lixin Gao, Weibo Gong, and Don Towsley. Monitoring and early warning for internet worms. In *10th ACM Conference on Computer and Communication Security (CCS 2003)*, October 2003.

[129] Cliff C. Zou, Weibo Gong, and Don Towsley. Worm propagation modeling and analysis under dynamic quarantine defense. In *ACM CCS Workshop on Rapid Malcode (WORM 2003)*, October 2003.

[130] Cliff Changchun Zou, Weibo Gong, and Don Towsley. Code Red worm propagation modeling and analysis. In *Proceedings of the 9th ACM Conference on Computer and Communication Security (CCS 2002)*, Washington, DC, November 2002.

Summary

English

This thesis introduces a new set of methods that offers a novel way of solving a broad class of problems involving process detection. Many of these problems have never been considered as processes, which is, in this authors opinion, why little actual progress has been made in solving them efficiently. With this thesis I hope to initiate a new way of thinking about problems as *processes*.

Almost everything that happens around us is process-like in nature. Consider, for instance, a car driving down a multi-lane highway. We can easily follow where the car is now, and where it is going. The car is in the *process* of driving, and we understand how the position of the car changes, based on its speed and direction. Essentially, we understand this process of moving. To formalize this concept a little bit, a process is usually considered to have some type of *state* that changes over time according to a set of laws (the process description). In case of the car, the state can be defined as its position, direction, and velocity. The process description, then, defines how this state changes over time, namely, if the car is traveling at 10 meters per second (the velocity) down the highway (the direction), its position will have changed to be 30 meters further down the road after three seconds have passed. Although this is a rough estimate, we do not need to be any more precise; after all, we only need to be accurate enough to realize the car is not going to hit us!

> → The beauty of process descriptions is that they are often elegant and straightforward, making them easy to understand.

Now imagine that we are driving on the same road with many cars around us. There will be cars passing, changing lanes, entering and exiting the highway, slowing down, and so forth. When we try to navigate on the road, we must keep a reasonably clear idea of what is happening near us, to avoid collisions or driving off the road. Unfortunately, we cannot look in all directions at once, meaning that we cannot keep a constant eye on all cars around us. We must therefore form a mental picture of where other cars are, and where they are going. What helps us greatly here is our understanding of the *process* of moving cars. For example, by looking in our rear-view mirror we see two cars behind us, both traveling in the same direction, one going a little faster than the other. What we have done now is take a partial sample of the environment, and learned the *state* of two cars, and our mental picture can be updated. As we drive along, we can now *predict* that one of those two cars may be passing us soon. This prediction helps us in making a decision not to move into the passing car's lane.

> → An important part of processes is that they allow us to make predictions on the future state of the environment.

One problem with our mental picture, however, is that it quickly becomes inaccurate. By simply looking at other cars, we cannot make very precise observations on speed, direction, and position. Therefore, we will need to look at them again soon, to correct for these inaccuracies. As a matter of fact, the longer we wait without looking at a car

that we know is there, the more we feel compelled to look at it. This way we constantly update and refine our mental picture of the environment around us. To formalize a little further, in our mental picture we have a certain number of *tracks*, each representing an instance of the process "moving car". Each time we look over our shoulder or in the mirror, we sample the state of other cars and match them with our predictions of where we expected them to be, basically updating our tracks. In general, we will take notice when a car has changed lanes, or when we did not see a car when we looked, because it was in a blind spot.

> → Using the predictions made by processes we can quickly assign new observations to existing tracks, even when these observations are in some way inaccurate.

Using our mental picture of the environment we are now able to quickly and safely make decisions in traffic. We can change lanes, enter or exit the highway, and pass other cars without colliding with them. This mental picture is often referred to as *situational awareness*, and the steps of updating this mental picture, based on what we observe and what we know about the processes that are happening, is at the core of a Process Query System, the topic of this thesis. A Process Query System lets a user describe processes that he or she is interested in, and the PQS then figures out if these processes are occurring in the observed environment. By using the steps described above, the PQS takes incoming observations and uses the described processes to form tracks of what events are happening, and reports those tracks back to the user. Since the PQS is a general engine, the user only needs to submit a description of the processes to build a full situational awareness system.

> → A Process Query System takes one or more process models and returns evidence of instances of these processes occurring in the observed environment.

This thesis describes the specifics of Process Query Systems, and demonstrates the power and versatility of the concept by applying PQS to several very diverse examples. For instance, in Chapter 4 a PQS is used in network security, where many different attacks are happening in large networks, while the network sensors only pick up partial data. In Chapter 3 a PQS is used to quickly identify new Internet worms, seconds after they are released. Then, in Chapter 5, a PQS is used for tracking fish swimming around in a tank, hiding behind obstacles and chasing each other. In that same Chapter, a PQS is used detect covert data exfiltration channels, and to monitor large server networks, making predictions about when services are about to fail. All of these examples have many processes occurring in a noisy, and lossy environment, where classical methods quickly break down. The power of a PQS comes from the fact that it only requires a process description to be submitted in order to track events, and have situational awareness in such difficult environments.

> → Process Query Systems can quickly and easily be applied to solve many practical problems.

Finally, this thesis only scratches the surface of what is possible with Process Query Systems. It lays bare an interesting new field of research in which many discoveries are yet to be made.

Dutch

Dit proefschrift introduceert een nieuwe groep methoden die een unieke manier bieden om een klasse van problemen te relateren aan process detectie, en ze daarmee op te lossen. Het merendeel van deze problemen zijn nooit in het kader van proces detectie bestudeerd. Dit is volgens de auteur van dit proefschrift de reden dat er weinig voortgang is geboekt in het efficiënt oplossen van deze problemen. Met dit proefschrift hoop ik een eerste stap te zetten tot het denken over problemen als processen.

Vrijwel alles wat rondom ons heen gebeurt is van nature proces georiënteerd. Neem bijvoorbeeld een auto die over een snelweg rijdt. We kunnen die auto gemakkelijk volgen, en voorspellen waar hij heen gaat. De auto is in het *proces* van rijden, en we begrijpen hoe de positie van de auto verandert, gebaseerd op zijn snelheid en richting. Of anders gezegd, wij begrijpen dit verplaatsingsprocess. Formeel gezegd, een proces is normaal gedefinieerd als een *staat* die met de tijd verandert volgens een aantal wetten (de procesbeschrijving). Voor de auto kunnen we de staat beschrijven als de positie, snelheid, en richting. De procesbeschrijving definieerd hoe deze staat veranderd met de tijd, namelijk, als de auto 10 meter per seconde (de snelheid) over de snelweg rijdt (de richting), dan is zijn positie na drie seconden 30 meter verder. Alhoewel dit een ruwe schatting is, hoeven we niet preciezer te zijn; immers, we hoeven slechts precies genoeg te zijn om te voorkomen dat de auto ons raakt!

> → Het mooie van procesbeschrijvingen is dat ze vaak elegant en eenvoudig zijn, waardoor ze gemakkelijk te begrijpen zijn.

Stel nu dat we op diezelfde snelweg rijden met vele andere auto's om ons heen. Er zullen auto's zijn die ons passeren, van baan veranderen, in- en uitvoegen, afremmen, enzovoorts. Als we in dit verkeer willen rijden moeten we dus een redelijk goed idee hebben van wat er om ons heen gebeurt om botsingen te voorkomen en niet van de weg te raken. Helaas kunnen we niet in alle richtingen tegelijk kijken, dus kunnen we niet altijd alle andere auto's direct zien. Daarom moeten we dus een mentaal beeld vormen van waar de andere auto's zijn, en waar ze heengaan. We worden hierbij geholpen door ons begrip van het proces van een rijdende auto. Bijvoorbeeld, als we in onze spiegel kijken, zien we twee auto's achter ons, alle twee rijden ze in dezelfde richting, ene een beetje sneller dan de andere. Wat we zojuist hebben gedaan is het nemen van een gedeeltelijke waarneming van onze omgeving. Deze gedeeltelijke waarneming vertelt ons de *staat* van twee auto's, en ons mentaal beeld kan nu worden bijgesteld. Als we blijven rijden kunnen we nu een *voorspelling* maken dat een van deze twee auto's ons spoedig zal passeren. Deze voorspelling helpt ons bij het besluit om niet van baan te veranderen.

> → Een belangrijk onderdeel van processen is dat ze ons in staat stellen om voorspellingen te maken over de toekomstige staat van onze omgeving.

Een probleem van ons mentaal omgevingsbeeld is dat het snel onnauwkeurig wordt. Door slechts naar andere auto's te kijken kunnen we niet erg precies meten wat hun snelheid, richting, en positie is. Daarom zullen we korte tijd later weer naar ze moeten kijken, om voor deze onnauwkeurigheden te corrigeren. Sterker nog, hoe langer we wachten zonder te kijken naar een auto waarvan we weten dat die er is, hoe sterker we ons gedwongen voelen om ernaar te kijken. Op deze manier verfijnen we constant ons mentaal beeld van de omgeving om ons heen. Om dit nog een stukje verder te formaliseren, in ons mentaal beeld hebben we een aantal *reeksen*, die ieder een instantie zijn van het proces "rijdende auto". Telkens als we over onze schouder of in de spiegel kijken, doen we een nieuwe waarneming van de andere auto's en vergelijken die met onze voorspellingen. Hierdoor kunnen we onze reeksen bijstellen en verbeteren. Over het algemeen merken we op wanneer een auto onverwachts van baan is veranderd, of als we een auto niet direct zichtbaar is vanwege een blinde hoek.

> → Door de voorspellingen van processen te gebruiken kunnen we snel nieuwe waarnemingen toevoegen aan bestaande reeksen, zelfs als deze waarnemingen onnauwkeurig zijn.

Door gebruik te maken van ons mentaal beeld van de omgeving kunnen we nu snel en veilig beslissingen nemen in het verkeer. We kunnen van baan veranderen, in- en uitvoegen, en andere auto's passeren zonder een botsing te veroorzaken. Dit mentale beeld wordt vaak beschreven als *omgevingsbewustheid*, en de stappen van het bijstellen van dit mentale beeld, gebaseerd op de waarnemingen en onze kennis van de processen in de omgeving, is de kern van Process Query Systems, het onderwerp van dit proefschrift. Een Process Query System stelt de gebruiker in staat processen te beschrijven waar hij of zij in geïnteresseerd is, en de PQS vindt dan uit of deze processen in de waargenomen omgeving voorkomen. Door gebruik te maken van de boven beschreven stappen, neemt de PQS binnenkomende waarnemingen, en gebruikt de procesbeschrijvingen om reeksen te vormen van de dingen die in de omgeving gebeuren. Deze reeksen worden dan vervolgens gerapporteerd aan de gebruiker. Omdat de PQS een algemene methode is, hoeft de gebruiker slechts één procesbescrijving te maken om een volledig omgevingsbewust systeem te bouwen.

> → Een Process Query System gebruikt één of meer proces beschrijvingen om bewijzen te vinden van de aanwezigheid van deze processen in de waargenomen omgeving.

Dit proefschrift geeft een precieze beschrijving van Process Query Systems, en demonstreert de kracht en breedheid van het concept door PQS toe te passen in een aantal zeer uiteenlopende voorbeelden. In hoofdstuk 4 wordt een PQS gebruikt voor netwerk beveiliging in grote computer netwerken, waar de sensoren slechts gedeeltelijke informatie verschaffen. In hoofdstuk 3 word een PQS toegepast om nieuwe Internet-wijde virussen te detecteren, slechts enkele seconden nadat ze zijn geactiveerd. Dan, in hoofdstuk 5, wordt een PQS gebruikt om vissen in een aquarium te volgen terwijl ze achter obstakels verdwijnen, en elkaar achterna zitten. In datzelfde hoofdstuk wordt een PQS

toegepast om verborgen communicatiekanalen te detecteren, en om grote server netwerken in de gaten te houden om te voorspellen wanneer servers op het punt staan uit te vallen. Alle bovenstaande voorbeelden hebben vele processen in omgevingen met veel storing en wegvallende waarnemingen. Klassieke methoden werken vaak niet goed in zulke omgevingen. De kracht van een PQS komt van het feit dat er slechts procesbeschrijvingen nodig zijn om gebeurtenissen te kunnen volgen, en omgevingsbewust te zijn, in zeer complexe omgevingen.

> → Process Query Systems kunnen snel en gemakkelijk worden toegepast om vele praktische problemen op te lossen.

Tenslotte, dit proefschrift beschrijft slechts het tipje van de ijsberg van wat mogelijk is met Process Query Systems. Het introduceert een interessant nieuw onderzoeksgebied, dat zich snel zal ontwikkelen.

Curriculum Vitæ

Vincent Berk was born on the 20^{th} of Februari 1978, in Leidschendam, the Netherlands. He lived most of his life in Zoeterwoude, finishing his VWO in 1996, after which he started his studies in Computer Science at Leiden University. At Leiden he was an active member of the student community, serving as the president of "de Leidsche Flesch", the student union for Mathematics, Physics, Astronomy, and Computer Science, for the year of 1998. He also held the student position on the board of the Faculty of Mathematics and Natural Sciences at Leiden University from May 1998, to September 1999. During his computer science studies he attended an intensive-program at the Technical University of Vienna, working on neural networks for image recognition. After finishing a research project on compiler optimizations for cache blocking, he moved to Hanover New Hampshire, USA, in August 2000, where he studied computer security at Dartmouth College while finishing his M.S. degree at Leiden. During this year he designed two new network security technologies that are still in use today: the "network-fuse", dynamically breaking connectivity for hosts showing hostile behavior, and the "URL-filter", which checks and validates every HTTP request before passing the request on to the actual web server. He also obtained a GIAC level-2 certification as a firewall and perimeter defense analyst. After obtaining his M.S. degree in June 2001, he started work as a full-time researcher and lecturer at the Thayer School of Engineering at Dartmouth College, working with prof. dr. G.V. Cybenko. There, he worked on his PhD. research in Internet-scale worm detection, which ultimately lead to the development of the Process Query System concept, guided by prof. dr. H.A.G. Wijshoff and prof. dr. G.V. Cybenko. He continues to live in New Hampshire, leading the development on Process Query Systems, and teaching the graduate Computer Architecure classes at Dartmouth College.