An Extensible Toolkit for Resource Prediction In Distributed Systems

Peter A. Dinda David R. O'Hallaron

July 1999
CMU-CS-99-138

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper describes the design, implementation, and performance of RPS, an extensible toolkit for building flexible on-line and off-line resource prediction systems in which resources are represented by independent, periodically sampled, scalar-valued measurement streams. RPS-based prediction systems predict future values of such streams from past values. Systems are composed at run-time out of an extensible set of communicating prediction components which are in turn constructed using RPS's sensor, prediction, and communication libraries. We have used RPS to evaluate predictive models and build on-line prediction systems for host load and network bandwidth. The overheads involved in such systems are quite low.

Effort sponsored in part by the Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0287, in part by the National Science Foundation under Grant CMS-9318163, and in part by a grant from the Intel Corporation. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

Keywords: resource prediction, performance prediction, linear time series models, time series prediction, prediction toolkits, distributed systems, predictive scheduling, application-level scheduling

1 Introduction

In order for a distributed, interactive application such as a scientific visualization tool [2] to be responsive to the user's demands when running on a shared, unreserved distributed computing environment, it must adapt its behavior to dynamically changing resource availability [8, 11]. Adaptation can take many forms, including dynamically choosing where to execute the application's tasks and changing the computation that a task performs, perhaps by changing the quality of its results.

Implicit in such application-level scheduling [5] is resource prediction. The application scheduler adapts the application's behavior based on predicted resource availability. Each resource has an associated sensor mechanism which periodically measures its availability as a scalar value. The prediction system for a particular resource uses an appropriate predictive model to map current and past resource measurements into predictions of future resource measurements. These predictions and the application's resource requirements are then mapped into an application-level metric such as task execution time. The scheduler uses these metrics to make scheduling decisions.

Because of their central role, resource prediction systems are of considerable importance. Unfortunately, tools to simplify studying resource prediction and building appropriate and efficient resource prediction systems are scarce. To remedy this situation, we have created the RPS (Resource Prediction System) toolkit. RPS is a set of libraries and programs implemented using them that simplifies the evaluation of prospective models and the creation of efficient and flexible resource prediction systems out of communicating components. This paper describes the design and implementation of RPS and the performance of a representative RPS-based resource prediction system.

To understand the role RPS plays, consider the process of building a prediction system for a new kind of resource. Once a sensor mechanism has been chosen, this entails essentially two steps. The first is an off-line process consisting of analyzing representative measurement traces, choosing candidate predictive models based on the analysis, and evaluating these models using the traces. The second step is to build an on-line prediction system that implements the most appropriate model with minimal overhead. There are a wide variety of statistical and signal processing tools for interactive analysis of measurement traces that work very well for performing most of the first step. However, tools for doing large scale trace-based model evaluation are usually ad hoc and do not take advantage of the available parallelism. With regard to the second step, building an on-line predictive system using the appropriate model, RPS provides tools for quickly building a on-line resource prediction system out of communicating components.

RPS is designed to be generic, extensible, distributable, portable, and efficient. The basic abstraction is the prediction of periodically sampled, scalar-valued measurement streams. Many such streams arise in a typical distributed system. RPS can be easily extended with new classes of predictive models and new components can be easily implemented using RPS. These components inherit the ability to run on any host in the network and can communicate in powerful ways. The only tool needed to build RPS is a C++ compiler, and it has been ported to four different Unix systems and Windows NT. For typical measurement stream sample rates and predictive models, the addition load that an RPS-based prediction system places on a host is in the noise, while the maximum sample rates possible on a typical machine are as high as 2.7 KHz.

Our experience shows that it is possible to use RPS to find and evaluate appropriate predictive models for a measure of resource availability, and then implement a low overhead prediction system that provides timely and useful predictions for that resource. We used RPS for an extensive evaluation of linear models for host load prediction [10] and a smaller evaluation of linear models for network bandwidth prediction. Following the evaluation studies, we used RPS to implement on-line prediction systems for these resources. These systems have been used in the CMU Remos [18] resource measurement system, the BBN QuO distributed object quality of service system [33], and are currently being integrated in to the Dv distributed visualization framework [2]. Parts of RPS have also been used to explore the relationship between SNMP network measurements and application-level bandwidth [19].

We begin our description of RPS by enumerating, in general terms, the steps that a researcher would follow in providing prediction for a new resource (Section 2.) Simplifying the final two steps of the process is the primary goal of RPS. In the same section, we also detail the additional requirements we placed on the RPS implementation. Next, we describe the high level composition of the RPS toolkit and how the pieces fit together (Section 3.) Then we describe each piece of RPS in detail: the sensor libraries, which measure host load and network flow bandwidth (Section 4); the time series prediction library which provides an extensible abstraction for prediction and implementations of a number of useful predictive models (Section 5); the mirror communication template library, which provides the infrastructure to build prediction components which can communicate in sophisticated ways (Section 6); and the prediction components we have built (Section 7.)

Throughout these sections, we provide examples of how these tools can be used by a researcher or practitioner. In Section 5.4, we describe the parallelized cross-validation system we implemented around the time series prediction library in order to evaluate prospective models on measurement traces.

We illustrate the performance of RPS in two ways. First, in Section 5.5, we detail the computational costs involved in using the various predictive models in the time series prediction library on representative measurement streams. The cost to fit and use different predictive models varies widely. Second, in Section 8, we describe a representative and realistic RPS-based prediction system for host load. We measure the performance of this system in terms of the achievable sample rates, the measurement to prediction latency, and the additional CPU and communication loads presented by the system when run at different sample rates. For samples rates that of are interest in this system, we find that the additional load is barely discernible. Its maximum achievable rate is about two orders of magnitude higher than we need.

Finally, we address related work in Section 9 and conclude in Section 10 with a discussion of the limitations of RPS and the work that is needed to simplify managing running RPS-based prediction systems.

2 Goal and requirements

The goal of RPS is to facilitate two aspects of the construction of on-line resource prediction systems. The aspects are the off-line evaluation of predictive models and the construction of on-line systems using appropriate models. In addition, we placed other requirements on RPS that we felt would provide flexibility for further research into resource prediction.

2.1 Designing a new prediction system

To understand the goal of RPS, it is useful to consider the task of a researcher interesting in predicting a new kind of resource. Roughly, he will follow these steps:

1. Construct a *sensor* for the resource. The sensor generates a periodically sampled, scalar-valued *measurement stream*. We will also refer to such a stream as a *signal*.

2. Collect measurement *traces* from representative environments.

3. Analyze these traces using various statistical tools.

4. Choose candidate models based on that analysis.

5. Evaluate the models in an unbiased off-line evaluation study based on the traces to find which of the candidate models are indeed appropriate.

6. Implement an on-line prediction system based on the appropriate models.

The first two steps generally require the researcher to implement custom tools using his domain-specific knowledge of the resource in question. When carrying out the third step, the researcher is aided by powerful, commonly available tools. This step, as well as the fourth step, also rely heavily on the statistical expertise of the researcher. The final two steps can benefit considerably from automated and reusable tools. However, such tools are scarce. The goal of RPS is to provide tools for these last two steps. The researcher should be able to use RPS to conduct off-line model evaluation, and then construct an on-line resource prediction system based on the appropriate models.

The first step requires domain-specific knowledge. There are a myriad of ways in which interesting signals in a distributed environment can be captured. For example, our load prediction work uses OS-specific mechanisms to retrieve Unix load averages (average run queue lengths) [9]. The Network Weather Service uses benchmarking to measure network bandwidths and latencies between hosts [30] and load average and accounting-based sensors for host load [31]. Remos uses SNMP queries to measure bandwidths and latencies on supported LANs [18]. These different mechanisms developed from the expertise of their researchers. Of course, some measurement issues are more general. For example, determining the signal's bandwidth and how to resample possibly aperiodically sampled signals to periodicity are important signal processing issues that arise for all signals.

Once the issues involved in the sensor are resolved, the researcher must use his expertise to choose a representative set of environments and capture traces from them using the sensor implementation. This second step is also highly dependent on the particulars of the signal and the environments of interest.

Unlike the first two steps, the third step and fourth steps, analyzing the collected traces and choosing candidate models based on the analysis, requires far less domain-specific knowledge and more general purpose statistical knowledge. Consequently, there are a number of tools available to help the researcher perform these steps. For example, our research into the properties of host load signals [9] made extensive use of the exploratory data analysis and time series analysis tools available in S-Plus [20], and in Matlab's [22] System Identification Toolbox [21].

Surprisingly, there are few tools to simplify the fifth step, evaluating the candidate models in an unbiased manner. Time series analysis methodologies such as that of Box and Jenkins [6], do generally have an evaluation step, but it is very interactive and primarily concerned with the fit of the model and not its predictive power, which is really what is of interest to the distributed computing community. Furthermore, these methodologies often assume that computational resources are scarce, when, in fact, they are currently widely available.

We believe that the appropriate way to to evaluate prospective models is to study their predictive performance when confronted by many randomly selected testcases. Running this plethora of testcases requires considerable computational resources. While it is possible to script tools such as Matlab and S-Plus to do the job, it would be considerably more efficient to have a more specialized tool that could exploit the embarrassing parallelism available here. Furthermore, it would be desirable for the tool to use the same model implementations that would ultimately be used in the on-line prediction system. We evaluated the use of linear models for host load prediction using such a parallelized tool implemented using RPS [10], which we describe in Section 5.4.

The final step, implementing an on-line prediction system for the signal, requires implementing, or reusing, the model or models that survived the evaluation step and enabling them to communicate with sensors and the applications or middleware that may be interested in the predictions. In addition, mechanisms to evaluate and control a prediction system must be provided. In Section 8.1 we show how we use RPS to implement an on-line host load prediction system using the same models we evaluated earlier.

2.2 Implementation requirements

The goal of RPS is to provide a toolkit that simplifies the final two steps of the prediction process, as described above. We also required that our implementation would provide generacity, extensibility, distributability, portability, and efficiency. These requirements were intended to make RPS as flexible as possible for future research into resource prediction. We address each of these requirements below, noting how our implementation addresses these requirements.

Generacity: Nothing in the system should be tied to a specific kind of signal or measurement approach. RPS should be able to operate on periodically sampled, scalar-valued measurement streams from any source. Generacity is important in a research system such as RPS because there are a plethora of different signals in a distributed system whose predictability we would like to study. While our research has focused primarily on the prediction of host load as measured by the load average, we have also used RPS to study the predictability of network bandwidth as measured by Remos and through commonly available traces.

Extensibility: It should be easy to add new models to RPS and to write new RPS-based prediction components. Being able to add new models is important because the statistical study of a signal can point to their appropriateness. For example, we added an implementation of the fractional ARIMA model to RPS after we noted that host load traces exhibited self-similarity. An obvious example of the need for easily constructable components is implementing new sensors. For example, we added a Remos-based network bandwidth sensor many months after writing a host load sensor.

Distributability: It should be possible to place RPS-based prediction components in different places on the network and have them communicate using various transports. On-line prediction

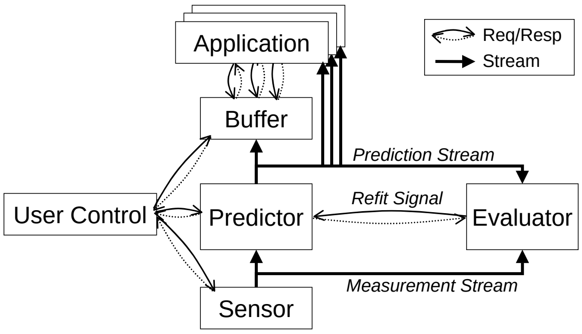


Figure 1: Overview of an on-line resource prediction system.

places computational load on the distributed system and it should be possible to distribute this load across the hosts as desired. Similarly, it should be possible to adjust the communication load and performance by using different transports. For example, many applications may be interested in host load predictions, so it should be easy to multicast them. If two communicating components are located on the same machine, they should be able to use a faster transport. RPS-based components are distributable and can communicate using TCP, UDP, Unix domain sockets, pipes, and files.

Portability: It should be easy to port RPS to new platforms, including those without threads, such as FreeBSD, and non-Unix systems, such as NT. FreeBSD and NetBSD are important target platforms for Remos and RPS. Some of the potential users of RPS such as organizations like the ARPA Quorum project, are increasingly interested in NT-based software. RPS requires only a modern C++ compiler to build and has been ported to Linux, Digital Unix, Solaris, FreeBSD, NetBSD, and Windows NT.

Efficiency: An on-line prediction system implemented using RPS components should be able to operate at reasonably high measurement rates, and place only minor computational and communication loads on the system when operating at typical sampling rates. Obviously, what is reasonable depends on the signal and the model being used to predict it. The idea here is not to achieve near-optimal performance, but rather to achieve sufficient performance to make an RPS-based resource prediction system usable in practice. Ultimately, something like a host load prediction system would probably be implemented as a single, hand-coded daemon which would be considerably faster than a system composed out of communicating RPS prediction components, such as we measure later. However, the latter RPS-based system can operate at over 700 Hz and offers noise-floor level load at appropriate rates with median prediction latencies in the 2 millisecond range. A comparable monolithic system, composed at compile-time using RPS, can sustain a rate of 2.7 KHz.

3 Overall system design

Here we describe the structure of RPS as it relates to the construction of an on-line resource prediction system (step 6 of Section 2.1.) In addition, RPS can also be used to implement off-line evaluation systems as per step 5 of Section 2.1. In the following, we focus on on-line prediction, pointing out the particulars of off-line prediction only in passing. Section 5.4 presents more details about a parallel off-line system.

Figure 1 presents an overview of an on-line time series prediction system. In the system, a *sensor* produces a *measurement stream* (we also refer to this as a *signal*) by periodically sampling some attribute of the distributed system and presenting it as a scalar. The measurement stream is the input of a *predictor*, which, for each individual measurement produces a vector-valued prediction. The vector contains predictions for the next m values of the measurement stream, where m is configurable. Each of the predictions in a vector is annotated with an estimate of its error. Consecutive vectors form a *prediction stream*, which is the output of the predictor. *Applications* (including other middleware) can subscribe directly to the prediction stream. The prediction stream also flows into a *buffer*, which keeps short history of the prediction stream and permits applications to access these predictions asynchronously, via a request/response mechanism. The measurement and prediction streams also feed an optional *evaluator*, which continuously monitors the performance of the predictor by comparing the predictor's actual prediction error with a maximum permitted error and by comparing the predictor's estimates of its error with another maximum permitted error level. If either maximum is exceeded — the predictor is either making too many errors or is mis-estimating its own error — the evaluator calls back to the predictor to tell it to refit its model. The user can exert control of the system by an asynchronous request/response mechanism. For example, he might change the sampling rate of the sensor, the model the predictor is using, or the size of the buffer's history.

The implementation of Figure 1 relies on several functionally distinct pieces of software: the sensor libraries, the time series prediction library, the mirror communication template library, the prediction components, and scripts and other ancillary codes.

Sensor libraries implement function calls that measure some underlying signal and return a scalar. Section 4 provides more information about host load and flow bandwidth libraries that we provide.

The **time series prediction library** provides an extensible, object-oriented C++ abstraction for time series prediction software as well as implementations of a variety of useful linear models. Section 5 provides a detailed description of this library and a study of the stand-alone performance of the various models we implemented.

The **mirror communication template library** provides C++ template classes that implement the communication represented by arrows in Figure 1. It makes it very easy to create a component, such as predictor, or any of the other boxes in the figure, which has a large amount of flexibility. In particular, the library provides run-time configurability, the ability to handle multiple data sources and targets, request/response interactions, and the ability to operate over a variety of transports. Section 6 describes the mirror library in detail.

Prediction components are programs that we implemented using the preceding libraries. They realize the boxes of Figure 1 and can be connected as desired when they are run. Section 7 describes the prediction components we implemented. Section 8 describes the performance and overhead of an on-line host load prediction system for composed from these components and communicating using TCP.

Ancillary software includes scripts to instantiate prediction systems on machines, tools for replaying host load traces, and tools for testing host load prediction-based schedulers. We don't describe this software in any further detail in this paper. One piece of software that has not been implemented is a system for keeping track of instantiated prediction systems and their underlying data streams. Currently, the client middleware or the user must instantiate prediction components and manage them.

4 Sensor libraries

Currently, two sensor libraries have been implemented. The first library, *GetLoadAvg* provides a function that retrieves the load averages (ie, average run queue length) of the Unix system it is running on. On some systems, such as Digital Unix, these are 5, 30, and 60 second averages, while on others, such as Linux, these are 1, 5, and 15 minute averages. We have shown elsewhere that the execution time of a compute-bound task on a Digital Unix system is strongly related to the average load it experiences during execution [10].

The GetLoadAvg code was borrowed from Xemacs and considerably modified. It uses efficient OS-specific mechanisms to retrieve the numbers where possible. When such mechanisms are not available, or when the user's permissions are inadequate, it runs the Unix uptime utility and parses its output. Because NT does not have an equivalent to the load average, it gracefully fails on that platform. Additional code is available from us for directly sampling the run-queue length on an NT system using the registry interface. On a 500 MHz Digital Unix machine, approximately 640,000 GetLoadAvg calls can be made per second. The maximum observed latency is about 10 milliseconds. We normally operate at about 1 call per second.

The second library, *GetFlowBW*, provides a function that measures the bandwidth that a prospective new flow between two IP addresses would receive, assuming no change in other flows. The implementation is based on Remos [18], which uses SNMP queries to estimate this value on LANs and benchmarking to estimate it on WANs. For SNMP queries on a private LAN, about 14 calls can be made per second.

5 Time series prediction library

The time series prediction library is an extensible set of C++ classes that cooperate to fit models to data, create predictors from fitted models, and then evaluate those predictors as they are used. While the abstractions of the library are designed to facilitate on-line prediction, we have also implemented several off-line prediction tools using the library, including a parallelized cross-validation tool. Currently, the library implements the Box-Jenkins linear time series models (AR, MA, ARMA, ARIMA), a fractionally integrated ARIMA model which is useful for modeling long-range dependence dependence such as arises from self-similar signals, a "last value" model, a windowed average model, and a long term average model. In addition, we implemented a template-based utility model which can be parameterized with another model resulting in a version of the underlying model that periodically refits itself to data.

5.1 Abstractions

The abstractions of the time series prediction library are illustrated in Figure 2. The user begins with a *measurement sequence*, $\langle z_{t-N}, \ldots, z_{t-2}, z_{t-1} \rangle$, which is a sequence of N scalar values that were collected at periodic intervals, and a *model template* which contains information about the structure of the desired model. Although the user can create a model template himself, a function

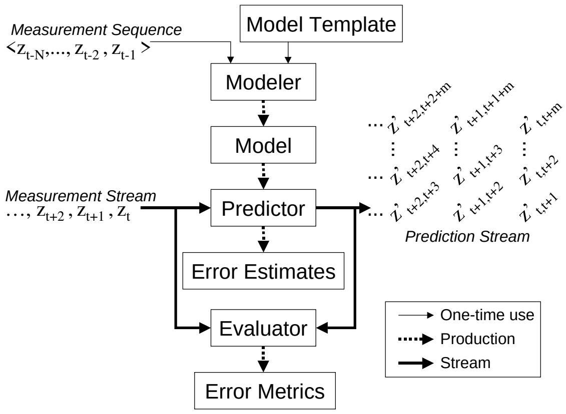


Figure 2: Abstractions of the time series prediction library

is also provided to create a model template by parsing a sequence of strings such as command-line arguments.

The measurement sequence and model template are supplied to a *modeler* which will fit a *model* of the appropriate structure to the sequence and return the model to the user. The user can select the appropriate modeler himself or use a provided function which chooses it based on the model template. The returned model represents a fit of the model structure described in the model template to the measurement sequence.

To predict future values, the model creates a *predictor.* A predictor is a filter which operates on a scalar-valued *measurement stream*, $z_t, z_{t+1}, \ldots$, producing a vector-valued *prediction stream*, $[\hat{z}_{t,t+1}, \hat{z}_{t,t+2}, \ldots, \hat{z}_{t,t+m}], [\hat{z}_{t+1,t+2}, \hat{z}_{t+1,t+3}, \ldots, \hat{z}_{t+1,t+1+m}], \ldots$ Each new measurement generates predictions for what the next m measurements will be, conditioned on the fitted model and on all the measurements up to and including the new measurement. m can be different for each step and the predictor can be asked for any arbitrary next m values at any point. The predictor can also produce *error estimates* for its $1, 2, \ldots, m$-step ahead predictions. Ideally, the prediction error will be normally distributed and so these estimates can serve to compute a confidence interval for the prediction.

The measurement and prediction streams can also be supplied to an *evaluator*, which evaluates the actual quality of the predictions independent of any particular predictor, producing *error metrics.* The user can compare the evaluator's error metrics and the predictor's error estimates to determine whether a new model needs to be fitted.

5.2 Implementation

The time series prediction library is implemented in C++. To extend the basic framework shown in Figure 2 to implement a new model, one creates subclasses of model template, modeler, model and predictor, and updates several helper functions. We implemented the nine models shown in Figure 3 in this way. Evaluator can also subclassed, but the base class already provides a comprehensive implementation.

In the remainder of this section, we shall focus on the implementation of the time series prediction library and the treatment of the individual modeling techniques is intentionally brief. More

Model	Notes
Simple Models	
MEAN	Long-range mean
LAST	Last-value
BM(p)	Mean over "best" window
Box-Jenkins Models	
AR(p)	Uses Yule-Walker
MA(q)	Uses Powell
ARMA(p,q)	Uses Powell
ARIMA(p,d,q)	Captures non-stationarity, uses Powell
Self-similar Models	
ARFIMA(p,d,q)	Captures long-range dependence
Utiltity Models	
REFIT<T>	Auto-refitting model

Figure 3: Currently implemented predictive models.

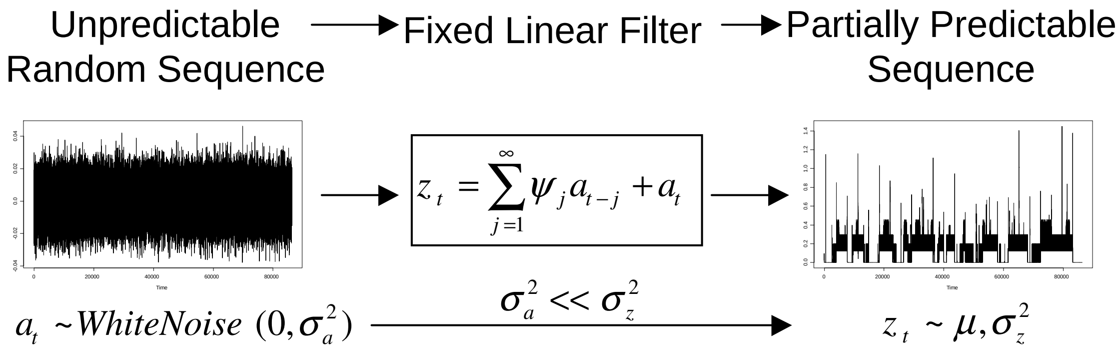


Figure 4: Linear time series model.

detail can be found in our earlier paper on linear models for host load prediction [10] and in standard references such as Box and Jenkins [6], Brockwell and Davis [7], and in the classic articles on fractional ARIMA models [17, 14]. S-Plus [20] and Matlab's System Identification Toolbox [21] provide good tools for learning and experimenting with these models.

The models we implemented are different kinds of linear time series models. In fitting such a model, the idea is to treat the measurement sequence $\langle z_t \rangle$ as the output of a linear filter being driven by a white noise sequence $\langle a_t \rangle$. Figure 4 illustrates this decomposition. The filter coefficients ψ_j are estimated from past observations of the sequence with the goal of minimizing the variance (or energy) of the driving source, σ_a^2. This residual variance is then interpreted as an estimate of prediction error of the model for one-step-ahead predictions.

This general form of the linear time series model is impractical, since it involves an infinite summation using an infinite number of completely independent weights. The practical models we implemented model the filter coefficients ψ_j as the coefficients of a ratio of polynomials in the backshift operator B, where $B^d z_t = z_{t-d}$. Using this scheme, the models we implemented are all variants of the following form:

$$z_t = \frac{\theta(B)}{\phi(B)(1-B)^d} a_t + \mu \tag{1}$$

5.2.1 Implemented models

MEAN model: The MEAN model has $z_t = \mu$, so all future values of the sequence are predicted to be the mean. This is the best predictor, in terms of minimum mean squared error, for a sequence which has no correlation over time — in other words, it is best if the sequence is entirely white noise. The MEAN modeler and model classes essentially do no work, while the MEAN predictor class maintains a running estimate of the mean and variance of the signal.

LAST model: LAST models have $z_t = \frac{1}{\phi(B)} a_t$ where $\phi(B)$ has one coefficient, set to one. In other words, $z_t = z_{t-1}$, so the one step ahead prediction is simply the last measured value. LAST is implemented as a BM(1) model, which we describe next.

BM(p) models: BM(p) models have $z_t = \frac{1}{\phi(B)} a_t$ where the $\phi(B)$ has N, $N \leq p$, coefficients, each set to $1/N$. This simply predicts the next sequence value to be the average of the previous N values, a simple windowed mean. The BM(p) modeler chooses N to minimize the one-step-ahead prediction error for the measurement sequence. The BM(p) model simply keeps track of this N and the BM(p) predictor implements the windowed average.

AR(p) models: AR(p) (purely autoregressive) models have $z_t = \frac{1}{\phi(B)} a_t + \mu$ where $\phi(B)$ has p coefficients which the modeler chooses to minimize σ_a^2. Our implementation uses the Yule-Walker technique to fit the model. In this technique, the autocorrelation function of the measurement sequence is computed to a maximum lag of p and then a p-wide Toeplitz system of linear equations in the coefficients is solved. Even for relatively large values of p, this can be done quite quickly, and the technique makes no assumptions about the error distribution. The AR(p) model stores the p coefficients, μ, and σ_a^2. Prediction is done with an Eta-theta predictor, which we describe next.

Eta-theta predictor: The AR, MA, ARMA, ARIMA, and ARFIMA models share a single predictor implementation. That predictor maintains a copy of the model coefficients ($\eta(B) = \phi(B)(1-B)^d$), $\theta(B)$ as before, μ, and σ_a^2. In addition, it maintains a prediction state in the form of a history of previous one-step-ahead predictions and their corresponding errors (the white noise). It operates linearly on this state and the coefficients to produce predictions. New measurement stream values change the state.

MA(q) models: MA(q) (purely moving average) models have $z_t = \theta(B) a_t$ where $\theta(B)$ has q coefficients. The modeler uses the Numerical Recipes implementation of Powell's method for multi-dimensional function minimization [24], pp. 406–413, to choose coefficients which minimize σ_a^2 for the measurement sequence. The MA(q) model stores the q coefficients, μ, and σ_a^2.

ARMA(p,q) models: ARMA(p,q) (autoregressive moving average) models have $z_t = \frac{\theta(B)}{\phi(B)} a_t + \mu$ where $\phi(B)$ has p coefficients and $\theta(B)$ has q coefficients. The modeler uses Powell's function minimization routine to choose the $p+q$ coefficients to minimize σ_a^2 for the measurement sequence. The ARMA(q) model stores the $p + q$ coefficients, μ, and σ_a^2.

ARIMA(p,d,q) models: ARIMA(p,d,q) (autoregressive integrated moving average) models implement Equation 1 for $d = 1, 2, \ldots$ The purpose of these unitary roots is to introduce integration of the signal, which allows ARIMA models to model non-stationary signals. The modeler fits

ARIMA(p,d,q) models by differencing the sequence d times and then fitting an ARMA(p,q) model as above to the result. The model contains the d, the $p+q$ coefficients, μ, and σ_a^2. When an eta-theta predictor is constructed, the d unitary roots are folded into the η portion of the predictor.

ARFIMA(p,d,q) models: ARFIMA(p,d,q) (autoregressive fractionally integrated moving average) models implement Equation 1 for fractional values of d, $0 < d < 0.5$. It can be shown that this fractional integration can model long-range dependence such as arises from self-similarity [4, 17, 14]. In addition, the "ARMA part" of the model models the short-range dependence in the signal. To fit ARFIMA models, we use Fraley's Fortran 77 code [12], which does maximum likelihood estimation of ARFIMA models assuming a normally distributed white noise source following Haslett and Raftery [16]. This implementation is also used by commercial packages such as S-Plus. When the predictor is constructed, we truncate $(1 - B)^d$ at 300 coefficients (other choices are possible).

REFIT<T> model: The REFIT<T> modeler, model, and predictor are C++ template classes that are parameterized by some modeler class and produce models of the underlying type that will automatically refit themselves at regular, user-specified, intervals. For example,

```
RefittingModeler<ARModeler>::Fit(seq,seqlen,modeltemplate,interval)
```

will return an AR model whose predictor will automatically fit a new AR model and update itself after every `interval` new samples.

5.2.2 Use of Powell's method

The choice of Powell's method, which we use in our implementations of the MA, ARMA and ARIMA models is a compromise. Powell's method does not require derivatives of the function being minimized, but operates more slowly than other methods which can make use of derivatives.

We use this method because we want to minimize σ_a^2 (the sum of squared prediction errors) directly. Other, faster methods to fit MA, ARMA, and ARIMA models exist. Instead of minimizing σ_a^2, these methods maximize the likelihood, which is a function of σ_a^2 whose form is determined by the distribution of the errors. By *assuming a particular distribution*, a function with known derivatives is produced and this allows the use of faster function minimization methods. However, we found that assuming a particular error distribution was rarely valid for host load and network flow bandwidth, two signals that were of considerable interest to us. The prediction errors of linear time series models on real signals are rarely distributed according to a convenient analytic distribution, although they usually are quite white (uncorrelated) and have low σ_a^2.

5.2.3 Prediction evaluation

The evaluator we implemented measures the following error metrics of a predictor. For each lead time, the minimum, median, maximum, mean, mean absolute, and mean squared prediction errors are computed. Of these, the mean squared prediction errors are especially useful, since they can be compared against the predictor's own estimates to determine whether a new model needs to be fitted. Of course, a new model can also be fitted if the prediction error is simply too high, or for any reason, at any time.

The one-step-ahead prediction errors (ie, a_{t+i}^1, $i = 1, 2, \ldots, n$) are also subject to IID and normality tests as described by Brockwell and Davis [7], pp. 34–37. IID tests include the fraction of the autocorrelations that are significant, the Portmanteau Q statistic (the power of the autocorrelation function), the turning point test, and the sign test. Recall that with an adequate model, the prediction errors should be uncorrelated (white) noise. If an IID test finds significant correlation in the errors, then a new model can be fitted to attempt to capture this correlation. The evaluator also tests if the errors are distributed normally by computing the R^2 value of a least-squares fit to a quantile-quantile plot of the errors versus a sequence of normals of the same mean and variance. If the R^2 is high, then using the simplifying assumption that the errors are normally distributed is well founded.

5.3 Example

The following is a code fragment to show how the time series prediction library can be used. In the code, we fit an ARMA(2,2) model to the first half of the sequence `seq` and then do 8-step-ahead predictions on the second half of the sequence

```
ModelTemplate *template;
Model         *model;
Predictor     *predictor;
Evaluator     *evaluator;
double        predictions[8], errorestimates[8];

// fit model to 1st half and create predictor
template  = ParseModel(3,{"ARMA","2","2"});
model     = FitThis(&(seq[0]),seqlen/2,*template);
predictor = model->MakePredictor();
eval      = new Evaluator;

// bring predictor state up to date
Prime(predictor,&(seq[0]),seqlen/2);

evaluator->Initialize(8);

// 8-ahead predictions for rest of sequecne
for (i=seqlen/2+1;i<seqlen;i++) {
  // Step the new observation into the predictor - this
  // returns the current one step ahead prediction, but
  // we're just ignoring it here.
  predictor->Step(seq[i]);
  // Ask for predictions + errors from 1 to 8 steps into the future
  // given the state in the predictor at this point
  predictor->Predict(8,predictions);
  predictor->ComputeVariances(8,errorestimates);
  // Send output to evaluator
  evaluator->Step(seq[i],predictions);
  // do something useful with  predictions here
}
```

```
// Get final stats from evaluator
evaluator->Drain();
PredictionStats *predstats = evaluator->GetStats();
```

To use a different model, all that is needed is to change the arguments to the `ParseModel` call, which could just as easily come from the command line. `ParseModel` and `FitThis` are helper functions to simplify dealing with the large and extensible set of available model templates, modelers, models, and predictors. It is also possible to invoke modelers directly, with or without model templates.

5.4 Parallel cross-validation system

Using the time series prediction library, we implemented a parallel cross-validation system for studying the predictive power of models one traces of measurement data. The user supplies a measurement trace and a file containing a sequence of testcase templates. A testcase template contains ranges of valid values for model classes, numbers of model parameters, lengths of sequences to fit models to, and lengths of subsequent sequences to test the fitted models on.

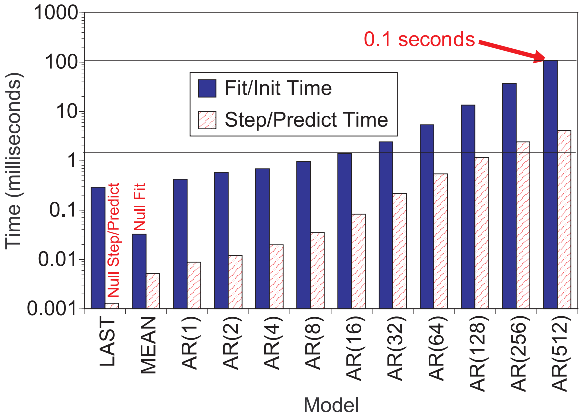
As the system runs, testcases are randomly generated by a master program using the template's limits on valid values and parceled out to worker processes using PVM [13]. The workers run code similar to that of Section 5.3 to evaluate a testcase. Essentially, the result is a set of error metrics for a randomly chosen model fit to a random section of the trace and tested on a subsequent random section of the trace. When a worker finishes evaluating a testcase, it sends the resulting set of error metrics back to the master, which prints them in a form suitable for importing into a database table for further study.

Because the testcases are randomly generated, the database of testcases can be used to draw unbiased conclusions about the absolute and relative performance of particular prediction models on particular kinds of measurement sequences.

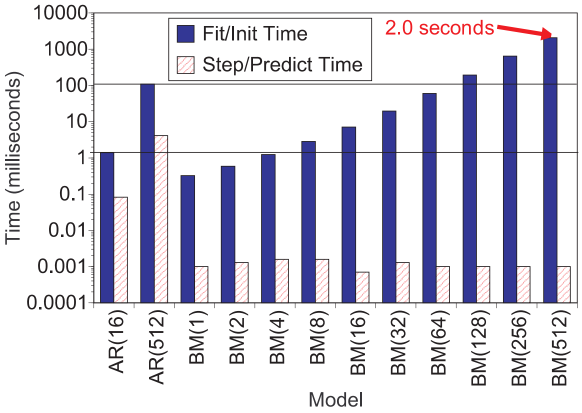
5.5 Performance

In implementing an on-line resource prediction system, it is obviously important to know the costs involved in using the various models supported by the time series prediction library. For example, if the measurement stream produces data at a 10 Hz rate and the predictor requires 200 ms to produce a prediction, then it will fall further and further behind, producing "predictions" for times that are increasingly further in the past. Clearly, such a predictor is useless. Another predictor that requires 100 ms will give up-to-date predictions, but at the cost of saturating the CPU of the machine where it is running. A predictor that requires 1 ms or less would clearly be desirable since it would consume only 1% of the CPU. Similarly, the cost of fitting a model and the measurement rate determines how often we can refit the model. At the 10 Hz rate, a model that takes 10 seconds to fit cannot be fit any more often than every 100 measurements, and only then if we are willing to saturate the CPU.

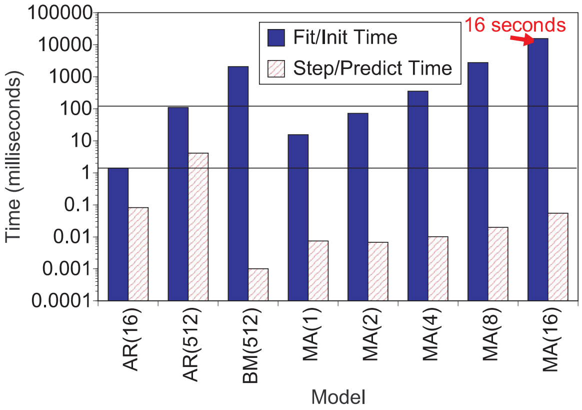
We measured the costs, in terms of system and user time required to (1) fit a model and create a predictor and (2) step one measurement into the predictor producing one set of 30-step-ahead predictions. Because the time to fit a model is dependent on the length of the measurement sequence, while the predictor time is not, we measured the costs for two different measurement sequence



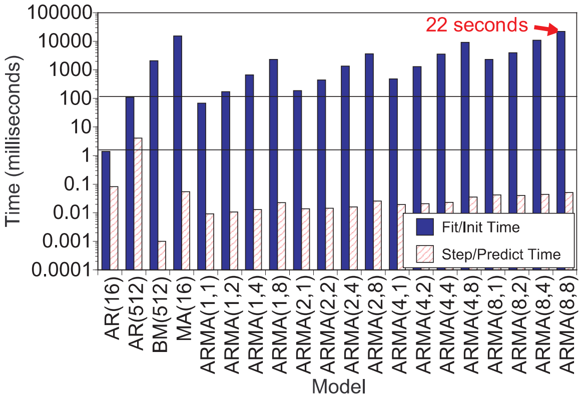
(a) AR Models



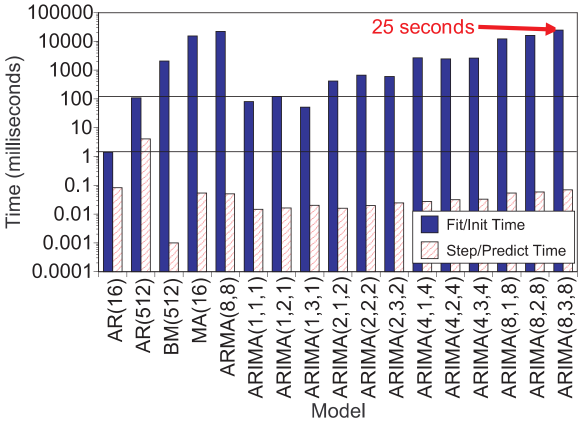
(b) BM Models



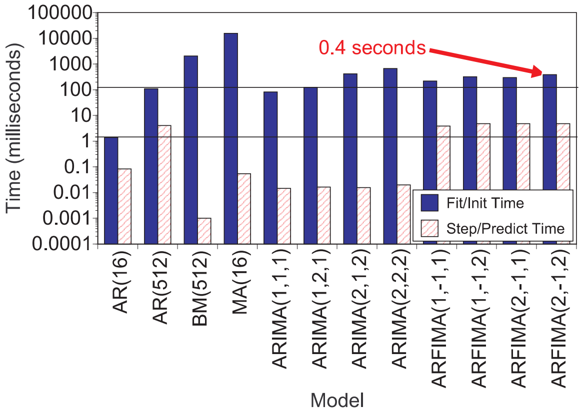
(c) MA Models



(d) ARMA Models



(e) ARIMA Models

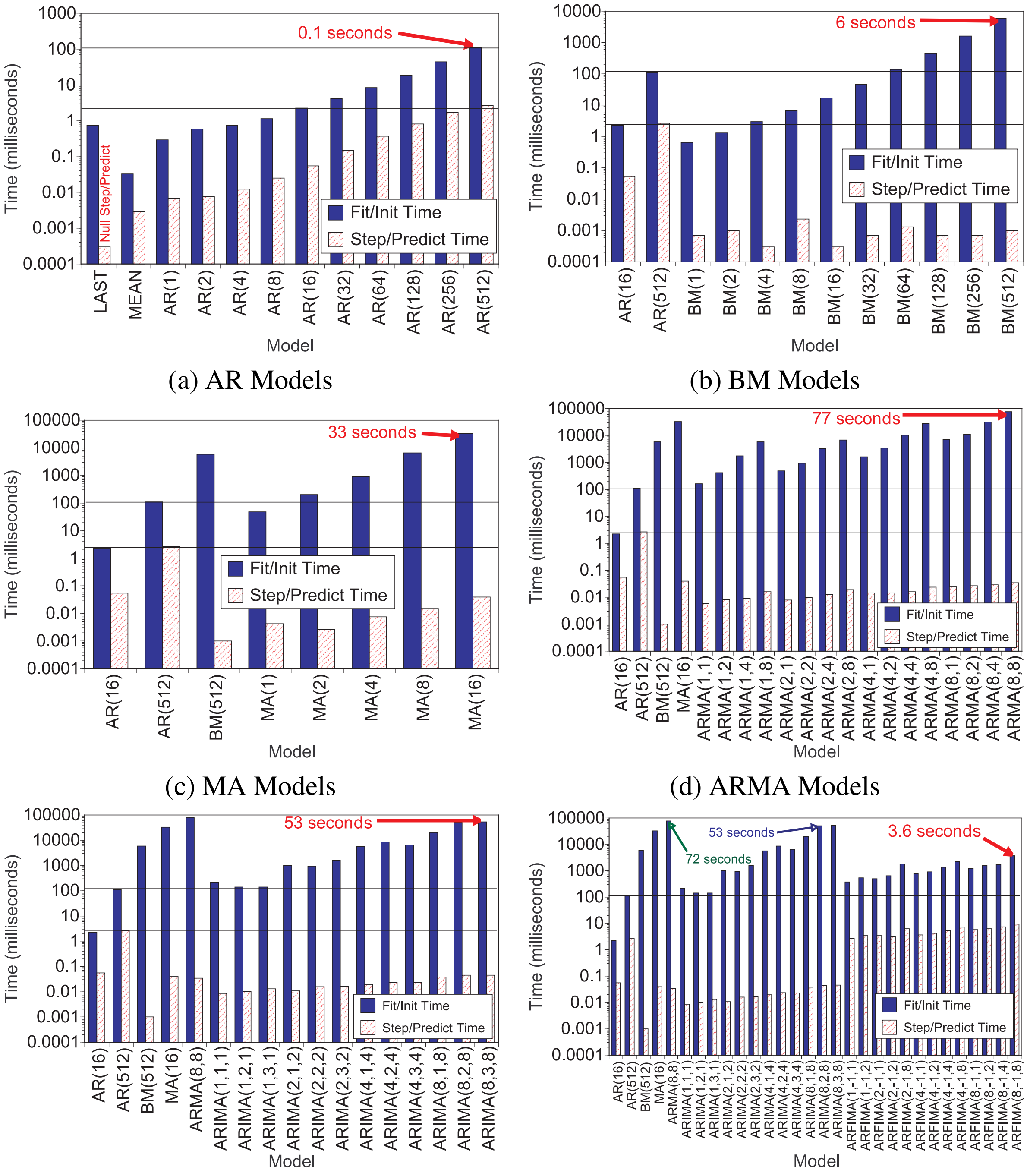


(f) ARFIMA Models

Figure 5: Timing of various prediction models, 600 sample fits.

lengths, 600 samples and 2000 samples. The measurement sequence used was a representative host load trace.

The results are shown in Figures 5 and 6. Each figure contains six plots, one for the (a) MEAN, LAST, and AR models, and one each for the remaining (b) BM, (c) MA, (d) ARMA, (e) ARIMA, and (f) ARFIMA models. The REFIT<T> variants were not measured, although their perfor-



(a) AR Models (b) BM Models

(c) MA Models (d) ARMA Models

(e) ARIMA Models (f) ARFIMA Models

Figure 6: Timing of various prediction models, 2000 sample fits.

mance can certainly be derived from the measurements we did take. For each model, we plot several different and interesting combinations of parameter values. For each combination, we plot two bars, the first bar (Fit/Init) plots the time to fit the model and produce a predictor while the second bar (Step/Predict) plots the time to step that predictor. Each bar is the average of 30 trials, each of which consists of one Fit/Init step and a large number of Step/Predict steps. The y axis on each plot is logarithmic. We replicate some of the bars from graph to graph to simplify comparing models across graphs and we also draw horizontal lines at roughly 1 ms and 100 ms, which are the Fit/Init times of AR(16) and AR(512) models, respectively. 1 ms is also the Step/Predict time of an AR(512) predictor.

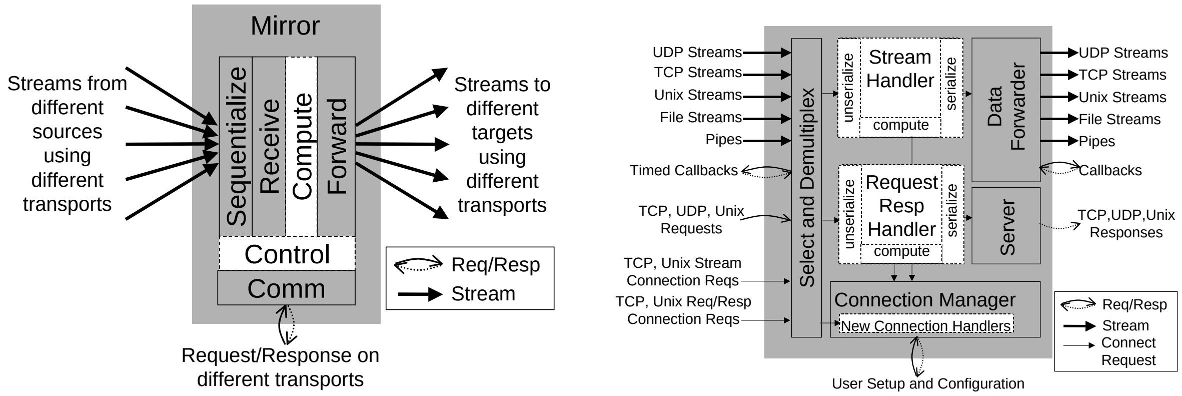
There are several important things to note when examining Figures 5 and 6. First, the inclusion of LAST and MEAN on the (a) plots provide measures of the overhead of the predictor and modeler abstractions, since LAST's predictor and MEAN's modeler do hardly any work. As we can see, the overhead of the abstractions are quite low and on par with virtual function calls, as we might expect.

The second important observation from the figures is that AR models, even with very high order, are quite inexpensive to fit. An AR(512) fit on a 2000 element sequence takes about 100 ms. In fact, ignoring LAST and MEAN, the only real competition to even the AR(512) comes from very low order versions of the other models. The downside of high order AR models is that the Step/Predict time tends to be much higher than that of lower order versions of the more complex models. For example, the predictor for an ARIMA(8,3,8) model operates in 1/100 the time of an AR(512). This is because the number of operations an eta-theta predictor performs is linear in the number of model parameters. If very high measurement rates are important, these more parsimonious models may be preferable. Interestingly, the ARFIMA models also have very expensive predictors. This is because, although the model captures long-range dependence very parsimoniously in the form of the d parameter, we multiply out the $(1 - B)^d$ term to generate 300 coefficients in the eta-theta predictor. It is not clear how to avoid this.

A final observation is that the MA, ARMA, and ARIMA models, quite surprisingly, are considerably more expensive to fit than the much more complex ARFIMA models. This is because we use a highly-tuned maximum likelihood code that assumes a normal error distribution to fit the ARFIMA model. The MA, ARMA, and ARIMA models are fit without making this assumption using a function optimizer which does not require derivatives. We used this approach because experimentation with Matlab, which uses a maximum likelihood approach, showed that the assumption was rarely valid for traces we were interested in. Maximum likelihood based modelers for MA, ARMA, and ARIMA models would reduce their Fit/Init times to a bit below those of the ARFIMA models. However, fitting even high-order AR models should still be cheaper because AR(p) models are fit by solving a p-diagonal Toeplitz while the other models require some form of function optimization over their parameters.

6 Mirror communication template library

As we began to implement an on-line resource prediction service for host load and contemplated implementing another for network bandwidth, we discovered that we were often rewriting the same communication code in each new program. As the number of such prediction components began to grow and we sought to incorporate more sophisticated communication transports such as multicast IP, the situation became untenable. Stepping back, we factored out the communication requirements of the prediction components and decided to implement support for them separately.



(a) (b)

Figure 7: The mirror abstraction (a) and implementation (b).

6.1 Motivation

Consider Figure 1, which shows a high level view of how the components of an on-line prediction service communicate. Notice that each component can is roughly similar in how it communicates with other components. It receives data from one or more input *data streams* and sends data to one or more output data streams. When a new data item becomes available on some input data stream, the component performs computation on it and forwards it to all of the output data streams. In addition to this data path the component also provides *request-response control* which operates asynchronously. We refer to this abstraction as a *mirror* (with no computation, input data is "reflected" to all of the outputs) and illustrate it in Figure 7(a).

We wanted to be able to implement the communication a mirror performs in different ways depending on where the components are situated and how many there are. For example, the predictor component in Figure 1 accepts a stream of measurements from a sensor and produces a stream of predictions which are consumed by the buffer and the evaluator. In addition, the evaluator and the user can asynchronously request that the predictor refit its model. Applications may connect at any time to receive the prediction stream. If we colocate all of the components on a single machine, Unix domain sockets or even pipes might be the fastest communication mechanism to use. If the components are on different machines — for example, it may be impossible to locate any components other than the sensor on a network router — TCP-based communication may be preferable. If a large number of applications are interested in the prediction stream, it may be necessary to use multicast IP in order to keep network traffic low. This is the kind of flexibility we expected from our mirror implementation.

6.2 Rejected implementation approaches

We considered five different ways of implementing the mirror abstraction. The first possibility was to implement each component as a CORBA [26] server and use remote method invocation (RMI) to transfer data for the streams as well as for control communication. This would have the advantage that the IDL compiler would generate all of the necessary object serialization code for us and simplify making changes in the future. However, it would require users of RPS to buy, install, and understand a CORBA implementation. Furthermore, there is no single COBRA implementation that is available on all the platforms we wanted to be able to port to, and some of our target platforms did not have a CORBA implementation at all.

The second possibility was to use an even higher level tool, the CORBA event channel service [26][196–204]. Event channels essentially provide the data path part of our mirror abstraction, but without computation. Multiple producers place atomic events into the channel and the channel delivers them to all listeners. An input event channel driving a single request/response server driving an output event channel would serve to implement a mirror. While this approach would have the greatly simplifying our implementation, it has the disadvantage that it would require users of RPS to not only buy, install, and understand a CORBA implementation but also an event channel implementation. Ultimately, we decided this would be too heavyweight.

The third possibility was to use Java. Java RMI [27] is in some ways even more powerful than CORBA RMI in that more complex object structures can be automatically serialized. Java also provides threads which would have considerably simplified programming mirrors. However, we would have had to make extensive use of the Java Native Interface (JNI) to call our prediction library and sensor code, both of which provide only C++ interfaces. Our experience with JNI on some of the more uncommon platforms we wanted to support, especially FreeBSD and NetBSD, led us to fear the Java approach. Avoiding JNI by porting the rest of RPS to Java to avoid was not really an option since some components, such sensors, must use native code.

The fourth possibility was to use our home-grown distributed object system, LDOS. LDOS consists of a CORBA IDL compiler and a lightweight run-time system. The combination allows for the easy development of C++ objects that can be called over the network via object reference stubs, a dynamic invocation mechanism, or via http and an html forms interface. Using LDOS would confer several of the benefits of the CORBA and Java approaches while not requiring users to purchase CORBA implementations or raising the complexity of JNI for us. However, LDOS relies heavily on native threads (pthreads or Win32 threads), which are simply not available on some of our target platforms, such as NetBSD and FreeBSD.

The final possibility we considered, and which we ultimately decided to implement, was a "from scratch" mirror implementation based on sockets and the Unix select call.

6.3 Implementation

Our mirror implementation, illustrated in Figure 7(b), is a C++ template class which is parameterized at compile-time by handlers for stream input and for request/response input. Additionally, it is parameterized by handlers for new connection arrivals for streams and for request/response traffic, although the default handlers are usually used for this functionality. Parameterized stream input and request-response handlers are also supplied for serializable objects, which can be used hide all the details of communication from the computation that a mirror performs for data or control. Beyond this, there are other template classes and default handler implementations to simplify using a mirror. For example, our prediction mirror implementation uses one of these templates, `FilterWithControl<>`, to simplify its design:

```
class Measurement : public SerializableInfo {...};
class PredictionResponse : public SerializableInfo {...};
class PredictionReconfigurationRequest : public SerializableInfo {...};
```

```
class PredictionReconfigurationReply : public SerializableInfo {...};

class Prediction {
...
public:
  static int Compute(Measurement &measure,
                     PredictionResponse &pred);
...
}

class Reconfiguration {
...
public:
 static int Compute(PredictionReconfigurationRequest &req,
                    PredictionReconfigurationResponse &resp);
...
}

typedef FilterWithControl<
  Measurement,
  Prediction,
  PredictionResponse,
  PredictionReconfigurationRequest,
  Reconfiguration,
  PredictionReconfigurationResponse
> PredictionMirror;
```

To implement serialization, the classes descended from `SerializeableInfo` implement methods for getting their packed size and for packing and unpacking their data to and from a buffer object. The implementer of the prediction mirror does not write any communication code, which is all provided in the `FilterWithControl` template which ultimately expands into a mirror template.

As shown in Figure 7(b), the heart of a mirror is a select statement that waits for activity on the file descriptors associated with the various input streams, request/response ports, and ports where new connections arrive. Streams can also originate from in-process sources, and so the select includes a timeout for periodically calling back to these local sources to get input stream data.

When the select falls through, all local callbacks that are past due are executed and their corresponding stream handler is executed on the new data item. Next, each open file descriptor that has a read pending on it is passed to its corresponding stream, request/response, or new connection handler. A stream handler will unserialize an input data item from the stream, perform computation on it yielding an output data item, which it passes to the mirror's data forwarder component.

The data forwarder will then serialize the item to all the open output streams. If a particular output stream is not writeable, it will buffer the write and register a handler with the selector to be called when the stream is once again writeable. This guarantees that the mirror's operation will not block due to an uncooperative communication target.

A request/response handler will unserialize the input data item from the file descriptor, perform computation yielding an output data item, and then serialize that output data item onto the same file descriptor. A new connection handler will simply accept the new connection, instantiate the appropriate handler for it (ie, stream or request response) and then register the handler with the connection manager.

The mirror class knows about a variety of different transport mechanisms, in particular, TCP, UDP (including multicast IP), Unix domain sockets, and pipes or file-like entities. The user asks the mirror to begin listening at a particular port for data or control messages either through an explicit mirror interface or by using `EndPoints`, which are objects that encapsulate all of a mirror's available transport mechanisms and can parse a string into an internal representation of a particular transport mechanism.

6.4 Example

Here is how the prediction server instantiates a prediction mirror that will receive measurements from a host named "pyramid" using TCP at port 5009, support reconfiguration requests via TCP at port 5010, and send predictions to all parties that connect via TCP at port 5011 or listen via multicast IP at address 239.99.99.99, port 5012, and also to standard out:

```
PredictionMirror mirror;
EndPoint tcpsource, tcpserver, tcpconnect
EndPoint multicasttarget, stdouttarget;

tcpsource.Parse("source:tcp:pyramid:5009");
tcpserver.Parse("server:tcp:5010");
tcpconnect.Parse("connect:tcp:5011");
multicasttarget.Parse("target:udp:239.99.99.99:5012");
stdouttarget.Parse("target:stdio:stdout");

mirror.AddEndPoint(tcpsource);
mirror.AddEndPoint(tcpserver);
mirror.AddEndPoint(tcpconnect);
mirror.AddEndPoint(multicasttarget);
mirror.AddEndPoint(stdouttarget);

mirror.Run();
```

In order to simplify writing clients for mirrors, we also implemented 3 reference classes, one for streaming input, one for streaming output, and one for request/response transactions. Here is how a client would begin to receive the multicasted prediction stream produced by the mirror code above:

```
EndPoint source;
StreamingInputReference<PredictionResponse> ref;
PredictionResponse pred;

source.Parse("source:udp:239.99.99.99:5012");
```

```
ref.ConnectTo(source);
while (...) {
   ref.GetNextItem(pred);
   pred.Print();
}
ref.Disconnect();
```

Similarly, here is the code that a client might use to reconfigure the prediction mirror via its TCP request/response interface, assuming the mirror is running on mojave:

```
EndPoint source;
Reference<PredictionReconfigurationRequest,
          PredictionReconfigurationResponse> ref;
PredictionReconfigurationRequest  req(...);
PredictionReconfigurationResponse resp;

source.Parse("source:tcp:mojave:5010");
ref.ConnectTo(source);
ref.Call(req,resp);
resp.Print();
```

7 Prediction components

Using the functionality implemented in the sensor libraries of Section 4, the time series prediction library of Section 5, and the mirror communication template library of Section 6, we implemented a set of *prediction components*. Each component is a program that implements a specific RPS function. On-line resource prediction systems are implemented by composing these components. The communication connectivity of a component is specified via command-line arguments, which means the location of the components and what transport any two components use to communicate can be determined at startup time. In addition, the components also support transient connections to allow run-time reconfiguration and to permit multiple applications to use their services. In Section 8.1 we compose an on-line host load prediction system out of the components we describe in this section.

We implemented a large set of prediction components, which are shown in Figure 8. They fit into five basic groups: host load measurement, flow bandwidth measurement, measurement management, stream-based prediction, and request/response prediction.

The host load measurement and flow bandwidth measurement groups implement sensors and tools for working with them. In each group, the sensor component (eg, loadserver, flowbwserver) generates a stream of sensor-specific measurements, while the other components provide mechanisms to control the sensor, read the measurement streams, buffer the measurement streams to provide asynchronous request/response access to the measurements, and, finally, to convert sensor-specific measurements into a generic measurement type. The remainder of the components use these generic measurement streams.

The measurement management group provides tools for receiving generic measurement streams, buffering generic measurements, and accessing such buffers.

Component	Function
	Host Load Measurement
loadserver	Generates stream of host load measurements
loadclient	Prints loadserver's stream
loadreconfig	Changes a loadserver's host load measurement rate
loadbuffer	Buffers measurements with request/response access
loadbufferclient	Provides access to a loadbuffer
load2measure	Converts a load measurement stream to generic measurement stream
	Flow Bandwidth Measurement
flowbwserver	Generates stream of flow bandwidth measurements using Remos
flowbwclient	Prints flowbwserver's stream
flowbwreconfig	Reconfigures a running flowbwserver
flowbwbuffer	Buffers a flow bandwidth measurement stream with request/response access
flowbwbufferclient	Provides access to a flowbwbuffer
flowbw2measure	Converts a flow bandwidth measurement stream to generic measurement stream
	Measurement Management
measureclient	Prints a generic measurement stream
measurebuffer	Buffers generic measurements with request/response access
measurebufferclient	Provides access to a measurebuffer
	Stream-based Prediction
predserver	Computes predictions for a generic measurement stream
predserver_core	Performs actual computations to contain failures
predreconfig	Reconfigures a running predserver
evalfit	Evalutes a running predserver and reconfigures it when necessary
predclient	Prints a prediction stream
predbuffer	Buffers a prediction stream with request/response access
predbufferclient	Provides access to a predbuffer
	Request/Response Prediction
pred_reqresp_server	Computes "one-off" predictions for request/response clients
pred_reqresp_client	Makes "one-off" prediction requests on a pred_reqresp_server

Figure 8: Prediction components implemented using RPS libraries.

The stream-oriented prediction group provides continuous prediction services for generic measurement streams. Predserver is the main component in this group. When started up, it retrieves a measurement sequence from a measurebuffer, fits the desired model to it, and then creates a predictor. As new measurements arrive in the stream, they are passed through the predictor to form m-step-ahead predictions and corresponding estimates of prediction error. These operations are similar to those described in Section 5.3. The actual work is done by a subprocess, predserver_core. This limits the impact of a crash caused by a bad model fit. If predserver_core crashes, predserver simply starts a new copy.

Predserver also provides a request/response control interface for changing the type of model, the length of the sequence to which the model is fit, and the number, m, of predictions it will make. This interface can be used by the user through the predreconfig program. Alternatively, and even at the same time evalfit can use the interface. Evalfit receives a generic measurement stream and a prediction stream, and continuously evaluates the quality of the predictions using an evaluator as discussed in Section 5.2.3. When the prediction quality exceeds limits set by the user, evalfit will force the predserver it is monitoring to refit the model.

The remaining components in the stream-oriented prediction services group simply provide buffering and client functionality for prediction streams.

The request/response prediction group provides classic client/server access to the time series prediction library. Pred_reqresp_client sends a measurement sequence and a model template to pred_reqresp_server, which fits a model and return predictions for the next m values of the sequence.

It is important to note that the set of of prediction components is not fixed. It is quite easy to construct new components using the libraries we described earlier. Indeed, we constructed additional components for the performance evaluation we describe in the next section.

8 Performance

The RPS-based prediction components described in the previous section are composed at startup time to form on-line prediction systems. To evaluate the performance of RPS for constructing such systems, we constructed a representative RPS-based prediction system and measured its performance in terms of the timeliness of its predictions, the maximum measurement rates that can be achieved, and the additional computational and communication load it places on the distributed system. In addition to the composed system, we also constructed a monolithic system using the RPS libraries directly and measured the maximum measurement rates it could support.

The conclusion of our study is that, for interesting measurement rates, both the composed and the monolithic systems can provide timely predictions using only tiny amounts of CPU time and network bandwidth. In addition, the maximum achievable measurement rates are 2 to 3 orders of magnitude higher than we currently need.

It is important to note that RPS is a toolkit for resource prediction, and, because of the inherent flexibility of such a design, it is difficult to measure RPS's performance for creating on-line resource prediction systems in a vacuum. Prediction components can be composed in many different ways to construct on-line resource prediction systems and the RPS libraries enable the construction of additional components or increased integration of functionality. Furthermore, prediction components can communicate in different ways. Finally, different resources require different measurement rates and predictive models. More complex predictive models require more computational resources while higher measurement rates require more computational and communication resources.

Because of the intractability of attempting to characterize this space, we instead focused on measuring the performance of a representative RPS-based on-line host load prediction system. The system is representative in the sense that it implements the functionality of Figure 1 using the prediction components. Furthermore, it is also a realistic system. It uses a predictive model that we have found appropriate for host load prediction in other work [10]. Finally, it is a fairly widely used system which has been distributed with Remos [18], is currently used in QuO [33], and is currently being used in our distributed real-time scheduling work [8].

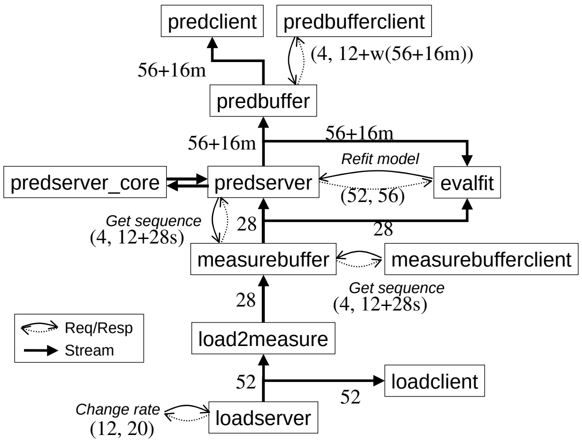


Figure 9: Online host load prediction system composed out of the RPS prediction components described in Section 7.

8.1 Host load prediction system

Figure 9 shows the configuration of prediction components used for host load prediction. The boxes in the figure represent prediction components while dark arrows represent stream communication between components, and symmetric arrows represent request/response communication between components. The arrows are annotated with communication volumes per cycle of operation of the system for streams and per call for request/response communication. s is the number of measurements being requested asynchronously from the measurebuffer while m is the number of steps ahead for which predictions are made and w is the number of predictions being requested from the predbuffer. Notice the similarity of this system to the high-level view of Figure 1.

The system works as follows. The loadserver component periodically measures the load on the host on which it is running using the GetLoadAvg library described in Section 4. Each new measurement is forwarded to any attached loadclients and also to load2measure, which converts it to a generic measurement form and forwards it to measurebuffer. Measurebuffer buffers the last N measurements and provides request/response access to them. It also forwards the current measurement to predserver and evalfit. Predserver consumes the measurement and produces an m-step-ahead prediction using its subprocess, predserver_core. It forwards the prediction to predbuffer and to evalfit. Evalfit continuously compares predserver's predictions with the measurements it receives from measurebuffer and computes its own assessment of the quality of the predictions. For each new measurement, it compares its assessment with the requirements the user has specified as well as with the predictor's own estimates of their quality. When quality limits are exceeded it calls predserver to refit the model. Predserver's predictions also flow to predbuffer, which provides request/response access to some number of previous predictions and also forwards the predictions to any attached predclients. Predbufferclients can asynchronously request predictions from predbuffer. Of course, applications can decide, at any time, to access the prediction stream or the buffered predictions in the manner of predclient and predbufferclient.

Each measurement that loadserver produces is timestamped. This timestamp is passed along as the measurement makes its way through the system and is joined with a timestamp for when the corresponding prediction is completed, and for when the prediction finally arrives at an attached predclient. We shall use these timestamps to measure the latency from when a measurement is made to when its corresponding prediction is available for applications.

The system can be controlled in various ways. For example, the user can change loadserver's measurement rate, the predictive model that predserver uses, and the time horizon for predictions. We used the control over loadserver's measurement rate to help determine the computational and communication resources the system uses.

So far, we have not specified where each of the components runs or how the components communicate. As we discussed in the previous section, RPS lets us defer these decisions until startup time and even run-time. In the study we describe in this section, we ran all of the components on the same machine and arrange for them to communicate using TCP. The machine we used is a 500 MHz Alpha 21164-based DEC personal workstation.

This configuration of prediction components is an interesting one to measure. It is reasonable to run all the components on a single machine since relatively low measurement rates and reasonably simple predictive models are sufficient for host load prediction [10]. It would be more efficient to use a local IPC mechanism such as pipes or Unix domain sockets to communicate between components. Indeed, a production host load prediction system might very well be implemented as a single process. We briefly discuss the performance of such an implementation in Section 8.3.3. TCP is interesting to look at because it gives us some idea of how well an RPS-based system might perform running on multiple hosts, which might be desirable, for, say, network bandwidth prediction. Furthermore, if RPS can achieve reasonable performance levels in such a flexible configuration, it is surely the case that a performance-optimized RPS-based system would do at least as well.

The predictive model that is used is an AR(16) fit to 600 samples and evalfit is configured so that model refitting does not occur. Predictions are made 30 steps into the future. The default measurement rate is 1 Hz. This model and rate is appropriate for host load prediction, as we discuss in earlier work [9, 10].

The following illustrates how the various prediction components are started:

```
% loadserver 1000000 server:tcp:5000 connect:tcp:5001 &
% loadclient source:tcp:`hostname`:5001 &
% load2measure 0 source:tcp:`hostname`:5001 connect:tcp:5002 &
% measurebuffer 1000 source:tcp:`hostname`:5002
   server:tcp:5003 connect:tcp:5004 &
% predserver source:tcp:`hostname`:5004
   source:tcp:`hostname`:5003 server:tcp:5005 connect:tcp:5006 &
% evalfit source:tcp:`hostname`:5004 source:tcp:`hostname`:5006
   source:tcp:`hostname`:5005
   30 999999999 1000.0 999999999 600 30 AR 16  &
% predbuffer 100 source:tcp:`hostname`:5006 server:tcp:5007
   connect:tcp:5008 &
% predclient source:tcp:`hostname`:5008  &
```

The use of measurebufferclient and predbufferclient are not shown above since these are run only intermittently.

8.2 Limits

Before we present the details of the performance of the host load prediction system, it is a good idea to understand the limits of achievable performance on this machine. Recall from Section 4 that the host load sensor library requires only about 1.6 μs to acquire a sample. As for the cost of prediction, Figure 5 indicates that fitting and initializing an AR(16) model on 600 data points requires about 1 ms of CPU time, with a step/predict time of about 100 μs. The computation involved in evalfit, load2measure, and the various buffers amounts to about 50 μs, thus the total computation time per cycle is 151.6 μs. If no communication was involved, we would expect the prediction system to operate at a rate no higher than 6.6 KHz.

However, the prediction system also performs communication. Examination of Figure 9 indicates that, for 30-step-ahead ($m = 30$) predictions, eight messages are sent for each cycle. There are 3 28 byte messages, 2 52 byte messages, and 3 536 byte messages. The measured bandwidths of the host for messages of this size are 2.4 MB/s (28 bytes), 4.2 MB/s (52 bytes), and 15.1 MB/s (536 bytes). Therefore the lower bound transfer times for these messages are 11.7 μs (28 bytes), 12.4 μs (52 bytes), and 35.5 μs (536 bytes). The total communication time per cycle is therefore at least $(3)11.7 + (2)12.4 + (3)35.5 = 166.4$ μs, and the total time per cycle is at least $151.6 + 166.4 = 318$ μs, which suggests a corresponding upper bound on system's rate of about 3.1 KHz.

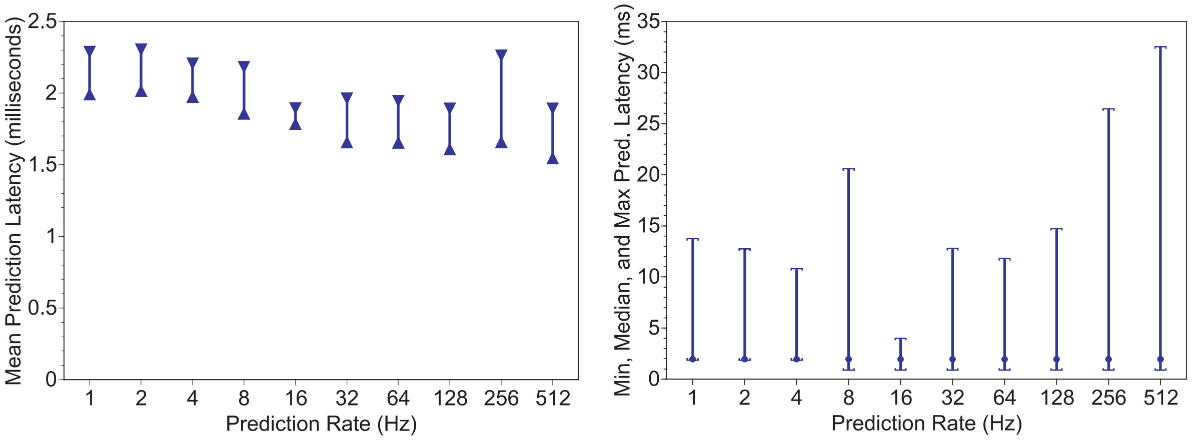
It is important to note that these rates are far in excess of the 1 Hz rate we expect from a host load prediction system, or even the 14 Hz peak rate for our Remos-based network sensor. What these high rates suggest, however, is that, for rates of interest to us, we can expect the prediction system to use only a tiny percentage of the CPU. In terms of communication load, we will only place $(3)28 + (2)52 + (3)536 = 1796$ bytes onto the network per cycle. At a rate of 14 Hz, this amounts to about 25 KB/s of traffic.

Of course, these are the upper limits of what is possible. We would expect that overheads of the mirror communication template library and the data copying implied by the TCP communication we use to result in lower performance levels.

The host we evaluated the system on has a timer interrupt rate of 1024 Hz, which means the all measurement rates in excess of this amount to "as fast as possible." This rate also results in a clock accuracy of approximately one millisecond.

8.3 Evaluation

We configured the host load prediction system so that the model will be fit only once, and thus measured the system in steady state. We measured the prediction latency, communication bandwidth, and the CPU load as functions of the measurement rate, which we swept from 1 Hz to 1024 Hz in powers of 2. We found that the host load prediction system can sustain measurement rates of 730 Hz with mean and median prediction latencies of around 2 ms. For measurement rates that are of interest to us, such as the 1 Hz rate for load and the 14 Hz for flow bandwidth, the additional load the system places on the machine is minimal.



(a) (b)

Figure 10: Prediction latency as a function of measurement rate: (a) 95% confidence interval of mean latency, (b) Minimum, median, maximum latency

8.3.1 Prediction latency

In an on-line prediction system, the timeliness of the predictions is paramount. No matter how good a prediction is, it is useless if if it does not arrive sufficiently earlier than the measurement it predicts. We measured this timeliness in the host load prediction system as the latency from when a measurement becomes available to when the prediction it generates becomes available to applications that are interested in it. This is the latency from the loadserver component to the predclient component in Figure 9.

The prediction latency should be independent of the measurement rate until the prediction system's computational or communication resource demands saturate the CPU or the network. Figure 10 shows that this is indeed the case. Figure 10(a) plots the 95% confidence interval for the mean prediction latency as a function of increasing measurement rates. We do not plot the latency for the 1024 Hz rate since at this point the CPU is saturated and the latency increases with backlogged predictions. Up to this point, the mean prediction latency is roughly 2 ms.

Figure 10(b) plots the minimum, median, and maximum prediction latencies as a function of increasing measurement rate. Once again, we have elided the 1024 Hz rate since latency begins to grow with backlog. The median latency is 2 ms, while the minimum latency is at 1 ms, which is the resolution of the timer we used. The highest latency we saw was 33 ms.

8.3.2 Resource usage

In addition to providing timely predictions, an on-line resource prediction system should also make minimal resource demands. After all, the purpose of the system is to predict resource availability for applications, not to consume the resources for itself.

To measure the CPU usage of our representative host load prediction system, we did the following. First, we started two resource monitors, vmstat and our own host load sensor. Vmstat is run every second and prints the percentage of the last second that has been charged to system and user time. Our host load sensor measures the average run queue length every second. After the

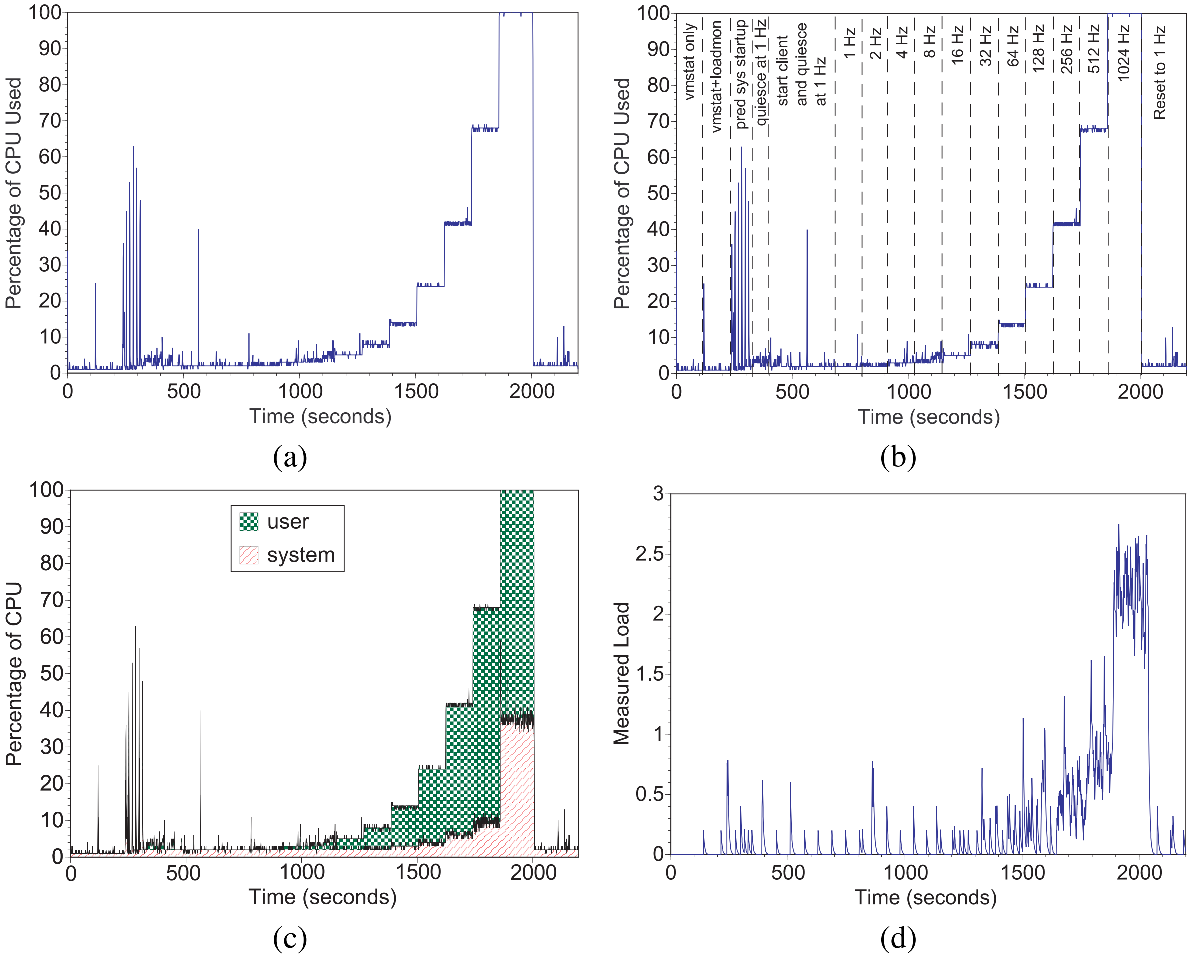


Figure 11: CPU load produced by system. The measurement rate is swept from 1 Hz to 1024 Hz. (a) shows total percentage of CPU used over time, (b) is the same as (a) but includes operational details, (c) shows user and system time, (d) shows load average.

sensors were started, we started the prediction system at its default rate of 1 Hz and let it quiesce. Next, we started a predclient and let the system quiesce. Then, we swept the measurement rate from 1 Hz to 1024 Hz in powers of 2. For each of the 10 rates, we let the system quiesce. Finally, we reset the rate to 1 Hz. Figure 11 shows plots of what the sensors recorded over time.

Figure 11(a) shows the percentage of the CPU that was in use over time, as measured by vmstat. Figure 11(b) is the same graph, annotated with the operational details described above. Figure 11(c) breaks down the CPU usage into its system and user components. The system component is essentially the time spent doing TCP-based IPC between the different components. Figure 11(d) shows the output of the load average sensor. When the load measured by this sensor exceeds one, we have saturated the CPU.

There are several important things to notice about Figure 11. First, we can sustain a measurement rate of between 512 Hz and 1024 Hz on this machine. Interpolating, it seems that we can sustain about a 730 Hz rate using TCP-based IPC, or about 850 Hz ignoring the system-side cost of IPC. While this is nowhere near the upper bound of 3.1 KHz that we arrived at in Section 8.2 , it

Rate (Hz)	Bytes/sec
1	1796
2	3592
4	7184
8	14368
16	28736
32	57472
64	114944
128	229888
256	459776
512	919552
1024	1839104

Figure 12: Bandwidth requirements as a function of measurement rate.

System	Transport	Optimal Rate	Measured Rate	Percent of Optimal
Monolithic	In-process	6.6 KHz	5.3 KHz	80 %
Monolithic	Unix domain socket	5.5 KHz	3.6 KHz	65 %
Monolithic	TCP	5.3 KHz	2.7 KHz	51 %
Composed	TCP	3.1 KHz	720 Hz	24 %

Figure 13: Maximum measurement rates achieved by monolithic and composed host load prediction systems.

is still much faster than we actually need for the purposes of host load prediction (1 Hz) and than the limits of our network flow bandwidth sensor (14 Hz).

In Section 8.3.3, we compare the maximum rate achievable by this composed host load prediction system to a monolithic system. The monolithic system achieves much higher rates overall, and those rates are closer to the upper bound.

A second observation is that for these interesting 1 and 14 Hz rates, CPU usage is quite low. At 1 Hz, it is around 2% while at 16 Hz (closest rate to 14 Hz) it is about 5%. For comparison, the "background" CPU usage measured when only running the vmstat probe is itself around 1.5%. Figure 11(d) shows that this is also the case when measured by load average.

Figure 12 shows the bandwidth requirements of the system at the different measurement rates. To understand how small these requirements are, consider a 1 Hz host load prediction system running on each host in the network and multicasting its predictions to each of the other hosts. Approximately 583 hosts could multicast their prediction streams in 1 MB/s of sustained traffic, with each host using only 0.5% of its CPU to run its prediction system. Alternatively, 42 network flows measured at the maximum rate could be predicted. If each host or flow only used the network to provided asynchronous request/response access to its predictions, many more hosts and flows could be predicted. For example, if prediction requests from applications arrived at a rate of one per host per second, introducing 552 bytes of traffic per prediction request/response transaction, 1900 hosts could operate in 1 MB/s.

8.3.3 A monolithic system

The composed host load prediction system we have described so far can operate at a rate 52–730 times higher than we need and uses negligible CPU and communication resources at the rates at which we actually desire to operate it. However, the maximum rate it can sustain is only 24% of the upper bound we determined in Section 8.2. To determine if higher rates are indeed possible, we implemented a monolithic, single process host load prediction system using the RPS libraries directly. This design can sustain a peak rate of 2.7 KHz when configured to use TCP, which is almost four times higher than the composed system.

Figure 13 shows the maximum rates the monolithic system achieved for three transports: in process, where the client is in the same process; Unix domain socket, where the (local) client listens to the prediction stream through a Unix domain socket; and TCP, where the client operates as with the earlier system. For comparison, it also includes the maximum rate of the composed system described earlier. In each case, we also show the optimal rate, which is derived in a manner similar to Section 8.2. The in-process case shows us the overhead of using the mirror communication template library, which enables considerable flexibility. That overhead is approximately 20%. The domain socket and TCP cases include additional, unmodeled overheads that are specific to these transports.

9 Related work

Application-level schedulers [5], such as best-effort real-time schedulers [8], are the primary users of on-line resource prediction systems.

Research into resource prediction has focused on determining appropriate predictive models for host behavior [10, 31, 25], and network behavior [30, 3, 15]. RPS is a toolbox that can help facilitate this research.

Interactive data analysis tools such as Matlab [22, 21] and S-Plus [20] provide many of the statistical and signal processing procedures needed to study resource signals and to find appropriate predictive models. Ideally, large scale, randomized evaluations of such candidate models are then needed. We have built RPS-based tools to help to efficiently conduct such studies.

Resource measurement systems, such as the Network Weather Service [32, 31, 30], Remos [18], and Topology-d [23] provide sensors that create the measurement streams that RPS-based systems can attempt to predict.

On-line resource prediction systems collect measurements from resource measurement systems and use them to predict future measurements. Other than RPS the Network Weather Service [32, 31, 30] (NWS) is the only example of an on-line resource prediction system we are aware of. While NWS is a production system that tries to provide a ubiquitous resource prediction service for metacomputing, RPS is a toolkit for constructing such systems and others. The RPS user can commit to as little or as much of RPS as is desired. NWS and RPS are complementary. For example, RPS-based systems could use NWS sensors, or NWS could use RPS's predictive models.

10 Conclusion

We have designed, implemented, and evaluated RPS, a toolkit for constructing on-line and off-line resource prediction systems in which resources are represented by independent, periodically sampled, scalar-valued measurement streams. RPS consists of resource sensor libraries, an extensive time series prediction library, a sophisticated communication library, and a set of prediction components out of which resource prediction systems can be readily composed. The performance of RPSis quite good. We measured a representative RPS-based host load prediction system and found that it provides timely predictions with minimal CPU and network load at reasonable measurement rates. The system can operate at measurement rates approaching 1 KHz, while a second, monolithic RPS-based system can operate at 2.7 KHz.

These results support the feasibility of resource prediction in general, and of using RPS-based systems for resource prediction in particular. The RPS-based host load prediction systems we implemented are being used in our research into prediction-based best-effort real-time systems [8] for interactive applications such as scientific visualization tools [2]. The systems have within the Remos measurement system [18] and QuO object quality of service system [33].

We have several extensions in mind for RPS. Currently, the user is responsible for instantiating and finding RPS-based prediction components. In the future, we intend to provide a service, probably over SLP [29], to manage running RPS components, prediction systems, and the measurement streams that they predict. We also will provide high-level calls to estimate task execution times based on application input and predicted host and network loads. We may increase the functionality of the mirror library's support for request/response communication, making it more like a CORBA ORB. This would make it possible to write template classes to make the construction of very fast monolithic prediction systems trivial. Finally, we will gradually add new predictive models as they become desirable. Currently, we are interested in providing support for nonlinear threshold autoregressive models [28] and models appropriate for chaotic signals [1].

Bibliography

[1] ABARBANEL, H. *Analysis of Observed Chaotic Data*. Institute for Nonlinear Science. Springer, 1996.

[2] AESCHLIMANN, M., DINDA, P., KALLIVOKAS, L., LOPEZ, J., LOWEKAMP, B., AND O'HALLARON, D. Preliminary report on the design of a framework for distributed visualization. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)* (Las Vegas, NV, June 1999).

[3] BASU, S., MUKHERJEE, A., AND KLIVANSKY, S. Time series models for internet traffic. Tech. Rep. GIT-CC-95-27, College of Computing, Georgia Institute of Technology, February 1995.

[4] BERAN, J. Statistical methods for data with long-range dependence. *Statistical Science 7*, 4 (1992), 404–427.

[5] BERMAN, F., AND WOLSKI, R. Scheduling from the perspective of the application. In *Proceedings of the Fifth IEEE Symposium on High Performance Distributed Computing HPDC96* (August 1996), pp. 100–111.

[6] BOX, G. E. P., JENKINS, G. M., AND REINSEL, G. *Time Series Analysis: Forecasting and Control*, 3rd ed. Prentice Hall, 1994.

[7] BROCKWELL, P. J., AND DAVIS, R. A. *Introduction to Time Series and Forecasting*. Springer-Verlag, 1996.

[8] DINDA, P., LOWEKAMP, B., KALLIVOKAS, L., AND O'HALLARON, D. The case for prediction-based best-effort real-time systems. In *Proc. of the 7th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 1999)*, vol. 1586 of *Lecture Notes in Computer Science*. Springer-Verlag, San Juan, PR, 1999, pp. 309–318. Extended version as CMU Technical Report CMU-CS-TR-98-174.

[9] DINDA, P. A. The statistical properties of host load. *Scientific Programming* (1999). To appear in fall of 1999. A version of this paper is also available as CMU Technical Report CMU-CS-TR-98-175. A much earlier version appears in LCR '98 and as CMU-CS-TR-98-143.

[10] DINDA, P. A., AND O'HALLARON, D. R. An evaluation of linear models for host load prediction. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC '99)* (August 1999). To Appear. Extended version available as CMU Technical Report CMU-CS-TR-98-148.

[11] FOSTER, I., AND KESSELMAN, C., Eds. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[12] FRALEY, C. Fracdiff: Maximum likelihood estimation of the parameters of a fractionally differenced ARIMA(p, d, q) model. Computer Program, 1991. http://www.stat.cmu.edu/general/fracdiff.

[13] GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHECK, R., AND SUNDERAM, V. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, Massachusetts, 1994.

[14] GRANGER, C. W. J., AND JOYEUX, R. An introduction to long-memory time series models and fractional differencing. *Journal of Time Series Analysis 1*, 1 (1980), 15–29.

[15] GROSCHWITZ, N. C., AND POLYZOS, G. C. A time series model of long-term NSFNET backbone traffic. In *Proceedings of the IEEE International Conference on Communications (ICC'94)* (May 1994), vol. 3, pp. 1400–4.

[16] HASLETT, J., AND RAFTERY, A. E. Space-time modelling with long-memory dependence: Assessing ireland's wind power resource. *Applied Statistics 38* (1989), 1–50.

[17] HOSKING, J. R. M. Fractional differencing. *Biometrika 68*, 1 (1981), 165–176.

[18] LOWEKAMP, B., MILLER, N., SUTHERLAND, D., GROSS, T., STEENKISTE, P., AND SUBHLOK, J. A resource monitoring system for network-aware applications. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing (HPDC)* (July 1998), IEEE, pp. 189–196.

[19] LOWEKAMP, B., O'HALLARON, D., AND GROSS, T. Direct queries for discovering network resource properties in a distributed environment. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC99)* (August 1999). To Appear.

[20] MATHSOFT, INC. *S-Plus User's Guide*, August 1997. See also http://www.mathsoft.com/splus.

[21] THE MATHWORKS, INC. *MATLAB System Identification Toolbox User's Guide*, 1996. see also http://www.mathworks.com/products/sysid.

[22] THE MATHWORKS, INC. *MATLAB User's Guide*, 1996. see also http://www.mathworks.com/products/matlab.

[23] OBRACZKA, K., AND GHEORGHIU, G. The performance of a service for network-aware applications. In *Proceedings of the ACM SIGMETRICS SPDT'98* (October 1997). (also available as USC CS Technical Report 97-660).

[24] PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. *Numerical Recipes in Fortran*. Cambridge University Press, 1986.

[25] SAMADANI, M., AND KALTHOFEN, E. On distributed scheduling using load prediction from past information. Abstracts published in Proceedings of the 14th annual ACM Symposium on the Principles of Distributed Computing (PODC'95, pp. 261) and in the Third Workshop on Languages, Compilers and Run-time Systems for Scalable Computers (LCR'95, pp. 317–320), 1996.

[26] SIEGAL, J. *CORBA Fundamentals and Programming*. John Wiley and Sons, Inc., 1996.

[27] SUN MICROSYSTEMS, INC. Java remote method invocation specification, 1997. Available via http://java.sun.com.

[28] TONG, H. *Threshold Models in Non-linear Time Series Analysis*. No. 21 in Lecture Notes in Statistics. Springer-Verlag, 1983.

[29] VEIZADES, J., GUTTMAN, E., PERKINS, C., AND KAPLAN, S. Service location protocol. Internet RFC 2165, June 1997.

[30] WOLSKI, R. Forecasting network performance to support dynamic scheduling using the network weather service. In *Proceedings of the 6th High-Performance Distributed Computing Conference (HPDC97)* (August 1997), pp. 316–325. extended version available as UCSD Technical Report TR-CS96-494.

[31] WOLSKI, R., SPRING, N., AND HAYES, J. Predicting the CPU availability of time-shared unix systems. In *Proceedings of the Eighth IEEE Symposium on High Performance Distributed Computing HPDC99* (August 1999), IEEE. To Appear. Earlier version available as UCSD Technical Report Number CS98-602.

[32] WOLSKI, R., SPRING, N. T., AND HAYES, J. The network weather service: A distributed resource performance forecasting system. *Journal of Future Generation Computing Systems* (1999). To appear. A version is also available as UC-San Diego technical report number TR-CS98-599.

[33] ZINKY, J. A., BAKKEN, D. E., AND SCHANTZ, R. E. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems 3*, 1 (April 1997).