Design, Implementation, and Performance of an Extensible Toolkit for Resource Prediction in Distributed Systems

Peter A. Dinda, *Member*, *IEEE*

Abstract—RPS is a publicly available toolkit that allows a practitioner to straightforwardly create flexible online and offline resource prediction systems in which resources are represented by independent, periodically sampled, scalar-valued measurement streams. The systems predict the future values of such streams from past values and are composed at runtime out of a large and extensible set of communicating components that are in turn constructed using RPS's extensible sensor, prediction, wavelet, and communication libraries. This paper describes the design, implementation, and performance of RPS. We have used RPS extensively to evaluate predictive models and build online prediction systems for host load, Windows performance data, and network bandwidth. The computation and communication overheads involved in such systems are quite low.

Index Terms—Distributed systems, performance of systems.

1 Introduction

Predicting the dynamic availability of resources such as CPU time [49], [11], network bandwidth [48], [37], and disk bandwidth is vital to adaptive applications [4], [1], load-balancing [21], [41], and others. 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, implementation, and performance of RPS.

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 offline 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 online 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 [32], [34], [33]. However, tools for doing large scale trace-based model evaluation are usually ad hoc and do not take advantage of the available parallelism. No tools exist to help with the second step, building an online predictive system using the appropriate model. RPS addresses both of these weak points.

The design goals of RPS were that it be generic, extensible, distributable, portable, and efficient. The basic abstraction is the prediction of periodically sampled, scalar-valued measurement streams, i.e., discrete-time signals. Many such streams arise in a typical distributed environment. RPS can be easily extended with new classes of predictive models and new components can be easily implemented. These components inherit the ability to run on any host in the network and can communicate in numerous ways. The only tool needed to build RPS is a C++ compiler and it has been ported to four different Unix systems and Microsoft Windows. For typical measurement stream sample rates and predictive models, the load placed on a host is miniscule, and sample rates, even on low-end machines, can exceed 2.7 KHz.

We have used RPS extensively to determine appropriate predictive models for host load [14], network bandwidth [37], and various Windows performance data [24], [25], and then to quickly implement low overhead prediction systems that provide timely and useful predictions for them. Some of these systems have been used in the CMU Remos [26], [28] resource measurement system, the BBN QuO distributed object quality of service system [51], and the Dv distributed visualization framework [1].

Unless it is stated otherwise, when the *user* is referred to, we mean the user of the RPS toolkit, not the end-user of a system built using RPS.

1.1 Related Work

Research into resource prediction has focused on determining appropriate predictive models for host behavior [14], [49], [38] and network behavior [48], [2], [18]. RPS is a toolbox that can help facilitate this research.

Resource measurement systems, such as the Network Weather Service [50], [49], [48], Remos [26], Topology-d

• *The author is with the Department of Electrical Engineering and Computer Science, Northwestern University, 2145 Sheridan Road, Evanston, IL 60208. E-mail: pdinda@cs.northwestern.edu.*

Manuscript received 26 July 2004; revised 12 Jan. 2005; accepted 23 Apr. 2005; published online 28 Dec. 2005.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0184-0704.

[35], Netlogger [20], SPAND [39], and others [29] provide sensors that create the measurement streams that RPS-based systems can attempt to predict.

Online resource prediction systems collect measurements from resource measurement systems and use them to predict future measurements. Other than RPS, the Network Weather Service [50], [49], [48] (NWS) is the only example of an online resource prediction system we are aware of. While NWS is a production system that tries to provide a ubiquitous resource prediction service for grid computing, 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 and this lets us explore the implementation space of prediction systems.

In terms of its available functionality compared to NWS, RPS includes a much wider range of predictive models, analysis tools, and communication mechanisms. RPS includes nonlinear models and a wavelet analysis toolkit and can even communicate using existing packets to reduce network overhead. RPS's computation and communication are structured so that an RPS-based prediction system can operate with very low overhead or at very high data rates.

2 Design Goals

At the highest level, RPS's goal is to simplify the creation of online measurement and prediction systems for resource availability. This process involves two steps, extensive offline evaluation using trace data, and the construction of an online system. It should be straightforward to move a model found appropriate during the offline step to online production use in the second. RPS serves the first goal through an extensive collection of libraries and command-line tools, including a parallel evaluation system. To serve the second goal, the same RPS models used offline can be used online and a prediction system can be quickly stitched together without programming. RPS was designed to meet the following requirements:

- **Genericity.** 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.
- **Extensibility.** It should be easy to add new models to RPS and to write new RPS-based components. Being able to add new models is important because the statistical study of a signal can point to them. Adding new sensors is necessary to measure new resources.
- **Distributed operation.** It should be possible to place RPS-based components in different places on the network and have them communicate using various transports. It should be possible to spread the computation and communication load of an RPS-based prediction system across multiple machines.
- **Portability.** It should be easy to port RPS to new platforms, including those without threads, such as FreeBSD, and non-Unix systems, such as Windows.
- **Efficiency.** An online prediction system must be able to operate at reasonably high measurement rates (100s of Hz) and place only minor computational and communication loads on the system when operating at typical sampling rates (1 Hz).

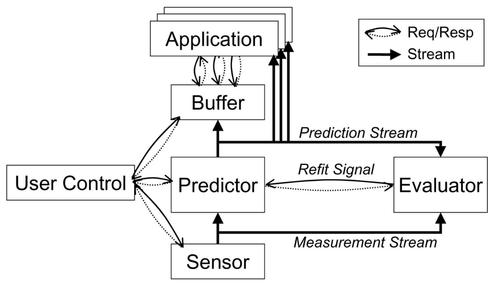

Fig. 1. Overview of an online resource prediction system.

3 System Design

In the following, we focus on online prediction, pointing out the particulars of offline prediction when needed. Fig. 1 presents an overview of an online 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 a 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 misestimating 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 Fig. 1 relies on several functionally distinct pieces of software: the sensors, the time series prediction library, the wavelet library, the communication library, the components, and scripts and other ancillary codes. A Web interface is also provided.

RPS has gone through three releases. The current release consists of $\sim$ 74,000 lines of C++, $\sim$ 21,000 lines of Perl, $\sim$ 6,900 lines of Fortran, and $\sim$ 600 lines of Java. The core elements of the system can be built and run using only a C++ compiler.

4 Sensors

A sensor measures some attribute of a resource, giving a scalar value. Currently, four sensors have been implemented. The first, *GetLoadAvg*, provides a library function that retrieves the load averages (i.e., average run queue length) of the Unix system it is running on. Where possible, it uses efficient OS-specific mechanisms to retrieve the measurements. On a 500 MHz Alpha 21164-based DEC personal workstation, it can run at a rate of 625 KHz.

The *GetFlowBW* library provides a function that measures the available bandwidth of a network path between two IP addresses. The implementation is based on Remos [27], which uses SNMP queries to estimate this value on LANs and benchmarking to estimate it on WANs. For SNMP queries on a private LAN, a rate of 14 Hz can be achieved.

The *WatchTower* sensor provides access to the performance counters on a Microsoft Windows computer. There are hundreds of counters that can be accessed using this library and more information is available elsewhere [25]. On a 500 MHz Pentium III machine, WatchTower can simultaneously monitor 256 performance counters at a rate of 16 Hz using 10 percent of the CPU while monitoring a single counter at 1 Hz requires less than 0.1 percent.

The *Proc* sensor provides access to performance data in the *proc* file system on Linux computers. The library includes a namespace for relevant counters and is responsible for reading and parsing them from proc. On a 1 GHz Pentium III computer running Red Hat 7.3, Proc can monitor a single counter at more than 1 KHz.

Other sensors can be readily added to the system simply by having them output a sequence of (optionally timestamped) scalars. A special component exists that can convert such a stream to RPS's internal representation and communicate it over the network.

5 Time Series Analysis and Prediction

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 online prediction, as per Fig. 1, RPS also includes offline prediction tools that use the library.

5.1 Abstractions

The abstractions of the time series library are illustrated in Fig. 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. The user can create a model template himself. Alternatively, he can use a provided function that creates an appropriate template by parsing a simple text-based specification of the template, given, for example, on a command line.

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 a modeler himself or use a provided function that chooses an appropriate modeler based on the model template. The returned model represents a fit of the model structure described in the model template to the measurement sequence.

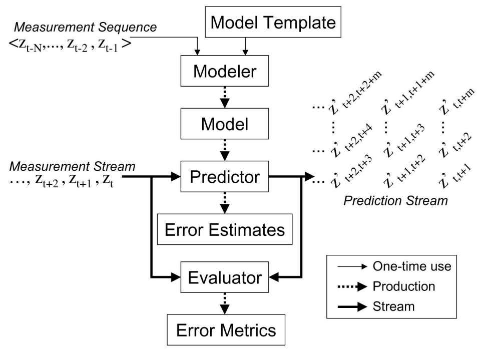

Fig. 2. Abstractions of the time series library.

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 independent 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.

It may appear that, since a model is fit and a predictor created only once, the cost of doing so is irrelevant. However, in practice, we often refit models based on the measured prediction errors of their predictors. The more frequently such refits occur, the more important the model fit cost becomes.

5.2 Implementation

The time series library is implemented in C++. To extend the basic framework shown in Fig. 2 to implement a new model, one creates subclasses of model template, modeler, model, and predictor, and updates several helper functions. The evaluator can also be subclassed, but the base class already provides a comprehensive implementation.

RPS implements a wide range of predictive models, as described below. Generally, these models fall into two classes: linear models and nonlinear models. Additionally, there are wrappers, which add functionality to models, and evaluators, which test models. The descriptions in the following are short. More detail can be found in our

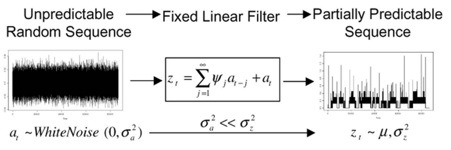
Unpredictable Random Sequence
Fixed Linear Filter
Partially Predictable Sequence
$z_t = \sum_{j=1}^{\infty} \psi_j a_{t-j} + a_t$
$a_t \sim WhiteNoise\ (0, \sigma_a^2)$
$\sigma_a^2 << \sigma_z^2$
$z_t \sim \mu, \sigma_z^2$

Fig. 3. Linear time series model.

experimental papers [14], [37], and the vast literature on time series analysis [6], [7], [23], [17], [46].

5.2.1 Linear Models

In fitting a linear 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 $\langle a_t \rangle$. Fig. 3 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 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:

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

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.

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.

$BM(p)$ 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.

$AR(p)$ (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. Even for relatively large values of p, this fitting process can be done quite quickly and the technique makes no assumptions about the error distribution.

$MA(q)$ (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 multidimensional function minimization [36 pp. 406-413], to choose coefficients which minimize σ_a^2 for the measurement sequence.

$ARMA(p,q)$ (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.

$ARIMA(p,d,q)$ (autoregressive integrated moving average) models implement (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 to the result.

ARFIMA(p,d,q) (autoregressive fractionally integrated moving average) models implement (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 [3], [23], [17]. 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 [15], which does maximum likelihood estimation of ARFIMA models assuming a normally distributed white noise source [22]. Modeling long-range dependence is necessary for modeling self-similar signals.

WAVELET models apply a streaming wavelet transform (see Section 6) to decompose the input signal into multiple detail signals. Each detail signal is predicted using some model, and these predictions inverse transformed to produce predictions of the input signal.

5.2.2 Nonlinear Models

There is no convenient framework for nonlinear models. In $BMED(p)$ models, the prediction is the median of the last p values in the input signal. The modeler chooses p to minimize the one-step-ahead prediction error for the measurement sequence. In $NEWTON(p)$) models, an interpolating polynomial of order p is fit to the last p values. It is then used to predict subsequent values. Any linear model can be converted into its nonlinear sibling using the MANAGED wrapper, described below.

5.2.3 Wrappers

Wrappers are C++ templates that can be wrapped around any other model to provide additional functionality while maintaining the model interface.

The AWAIT wrapper introduces a delay in the predictor produced by a modeler. It requires that it receive a certain number of input samples before the predictor is engaged to produce outputs. Sample delay is needed in various places, such as in the WAVELET model.

The REFIT wrapper causes the underlying model to be refitted at regular user-specified intervals.

The MANAGED wrapper is a large and very important wrapper that does continuous evaluation of a predictor and forces its model to be refit if error limits are exceeded. This has the effect, when used with a linear model, of turning it into its nonlinear "threshold" analogue [46]. This is a very useful wrapper because many signals generated by resources are not stationary as assumed by most models, but, rather, have stationary periods [9]. We have used MANAGED wrappers extensively in our work.

Both the MANAGED and REFIT wrappers cause their underlying models to be refit, increasing the importance of their fit cost.

5.2.4 Evaluator

RPS's evaluator implementation measures the following error metrics of a predictor. For each lead time (number of steps ahead), 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 (i.e., a^1_{t+i}, $i = 1, 2, \ldots, n$) 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 Code Size

The interface to the time series library is relatively straightforward, resulting in the user writing a small amount of code to use the library. A set of utility functions are included that make it possible to use the library generically, allowing the choice of model to be determined from a runtime string. This makes it possible to extend the range of models provided without modifying code that uses the library. It is possible to use all elements of Fig. 2 in this way in about 20 lines of C++, as shown elsewhere [13].

5.4 Parallel Evaluation System

RPS includes a program that uses master/slave parallelism implemented using PVM [16] to do large-scale randomized evaluations of predictive models on trace data. The user supplies a measurement trace and a file containing a sequence of test-case templates that specify models to examine and ranges of parameters. As the system runs, test cases containing specific models and segments of the trace are randomly generated by the master using the template. When a slave evaluates a test case, 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. These test case parameters and results are then imported into a database. Because the test cases are randomly generated, the database of test cases can be used to draw unbiased conclusions about the absolute and relative performance of particular prediction models on particular kinds of measurement sequences, as we have done for host load [14] and network bandwidth [37].

5.5 Runtime Costs

In an online prediction system, it is vital to understand the runtime cost of a predictive model. This cost in part determines the maximum rate at which the system can be run, the delay before the prediction is ready, and the overhead the system will have when run at a lower rate. In our original use of RPS, host load prediction, the goal was that the predictor would have a delay of 1 ms and consume only 1 percent of the CPU when run at 1 Hz.

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. We show both fit and step costs because, in typical use, models are often refit when prediction quality declines, either explicitly or via a mechanism like the REFIT and MANAGED wrappers. As refitting becomes more frequent, the fit cost becomes more important. In general, there is a trade-off between fit cost and step cost. The more complex the structure of the model, the higher the fit cost, but the lower the step cost.

For all of the models we implement, the step cost is independent of the data in the measurement stream. However, the fit cost depends on both the length of the measurement sequence (generally between linearly and quadratically) in all cases and on the nature of the data in some cases. In practice, fitting is usually dominated by computing the ACF or PACF, which can be done in $O(NlogN)$ time. When building a system, the cost of a particular model must also be ameliorated by its prediction accuracy. We summarize the prediction accuracy of our various models on host and network data in Section 10. Here, we simply show a wide range of models, looking at their fit and step costs averaged over randomly selected segments of a representative host load trace. For host load prediction, AR(16) or better predictive models are appropriate, typically combined with a MANAGED wrapper, giving us a simple threshold autoregressive model.

We measured the fit and step costs for two different measurement sequence lengths, 600 samples and 2,000 samples. For space reasons, we show only the 600-sample case here, which is the same case used in describing the online host load prediction system in Section 9. The machine used is a 500 MHz Alpha 21164-based DEC personal workstation.

Fig. 4 shows the result. There are 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. 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. One ms is also the Step/Predict time of an AR(512) predictor.

The WAVELET, BMED, and NEWTON models are not shown here. The times for BMED and NEWTON are similar to those of BM. The time for an l-$level$ wavelet predictor with the same model at each level is typically $l \times t_{model}$, where t_{model} is the time for the model.

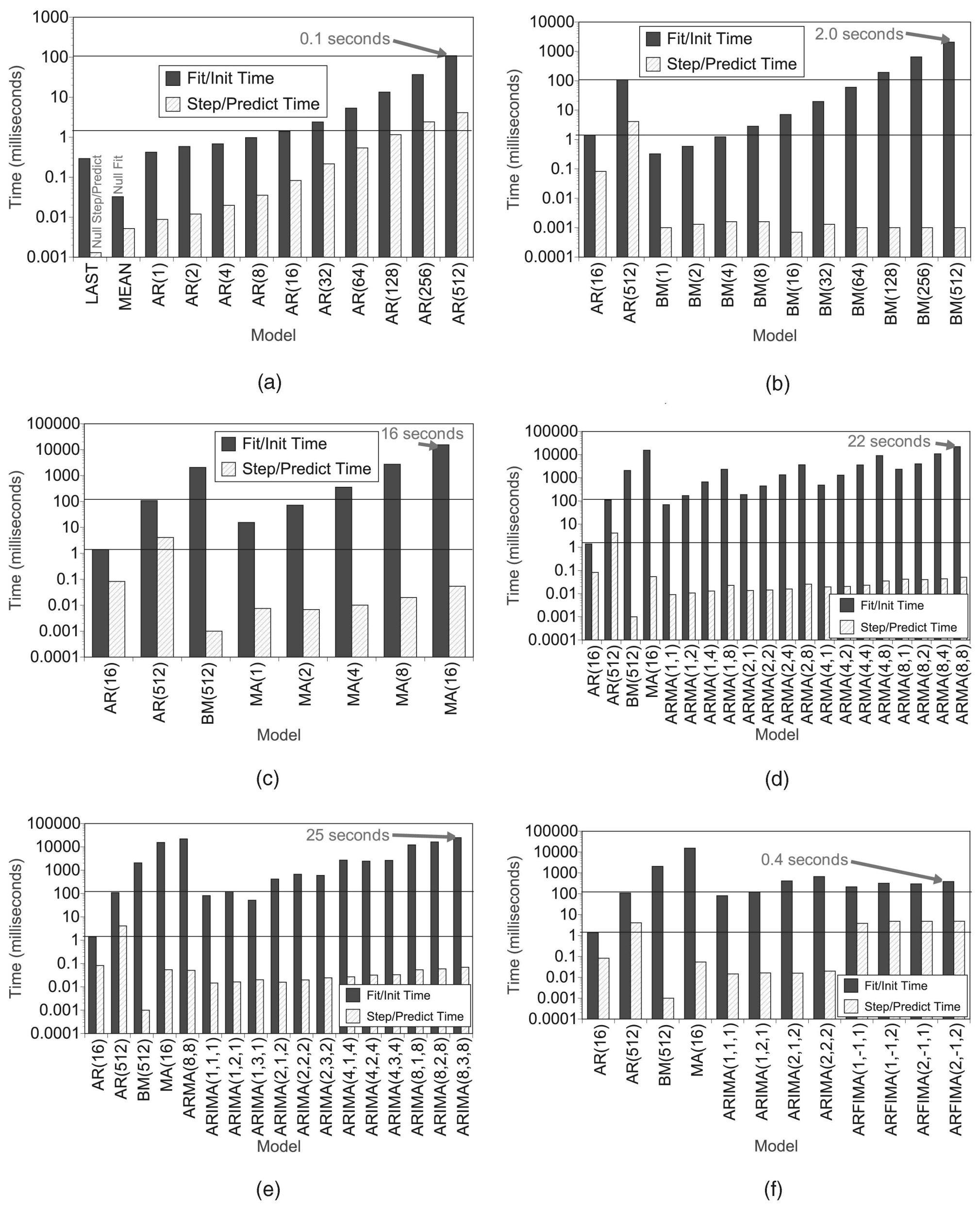

Fig. 4. Timing of various prediction models, 600 sample fits. (a) AR models. (b) BM models. (c) MA models. (d) ARMA models. (e) ARIMA models. (f) ARFIMA models.

If we consider the times to fit and use the simplest models, like LAST and MEAN, we can see that the overhead of our abstraction is quite low and on par with a few virtual function calls, as we might expect. The $AR(p)$) model combines low and deterministic fit and use overheads with statistical rigor. For this reason, it is often our model of choice, even if another model can do the job more parsimoniously. However, it is not always the most appropriate or fastest model.

Notice that the time to fit and use the models varies over a tremendous range, several orders of magnitude. Furthermore, because the fitting of many of the models involves search, the time can be data dependent. This is one of the core motivations behind RPS's communication tools and componentization, as described in Sections 7 and 8. Using these elements of RPS, we can place sensors, prediction, and evaluation in different locations.

6 Wavelet Analysis

RPS includes a wavelet toolkit written expressly for use in distributed systems. Like the time series library, the wavelet library supports both offline analysis of signals and the construction and deployment of online systems.

A wavelet transform converts a periodically sampled, time domain signal, such as provided by a sensor, into two dimensions representing time and frequency. The outputs of the wavelet transform are called the wavelet coefficients (or "details") and can be studied in lieu of the original signal for they contain all the information in it. The wavelet domain exposes opportunities that do not exist in time

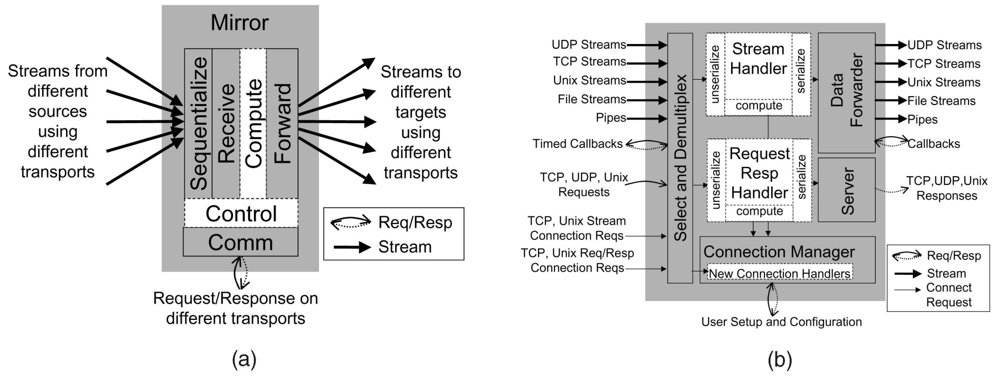

Fig. 5. (a) The mirror abstraction and (b) implementation.

domain or frequency domain and a wavelet domain signal can be readily converted back to time domain without loss of information.

Many implementations of wavelet analysis exist. However, none are tuned for resource monitoring in distributed systems, where we require an inexpensive streaming operation at low sample rates and efficient communication over lossy channels. In addition, few provide generality, allowing the user to construct essentially arbitrary transforms in pursuit of research goals. Using the primitives included in our wavelet toolkit, it is possible to build arbitrary transforms and adaptively shape them at runtime. The toolkit is written in C++ using templates and, so, can be instantiated for many different input and output data types and different wavelet basis functions.

Building on these primitives, the library includes implementations of discrete-time transforms and streaming transforms, both sample-based and sample block-based. Variants of the streaming transforms that support dynamic reconfiguration at runtime are provided. Finally, all transforms can be run in a "mixed" mode, meaning that they can provide any combination of approximations and details as output or use them as input for the inverse transforms. This provides support for straightforward multiresolution analysis.

On a 2 GHz Pentium 4 machine, a typical 10-level streaming forward transform based on a D8 wavelet and using double precision floating-point can operate at a sample rate of 156 KHz. The transform can run at 40 KHz with less than 10 percent CPU utilization. Similar to the time series library, common uses of the wavelet library, running at typical sample rates for resource monitoring, will have miniscule costs and impact on other work on the machine. More detailed information about the wavelet toolkit can be found in a technical report [43]. We discuss the use of the wavelet toolkit for prediction in Section 10.

7 Communication

In any prediction system, measurements, predictions, evaluation results, and control information must be conveyed from producers to consumers. Communication should be as efficient as possible, since RPS traffic on the network consumes bandwidth that the application traffic can use. When configuring a set of communicating RPS components, we want to be able to make use of the most efficient communication transport possible without reprogramming.

7.1 Abstraction

Consider Fig. 1, which shows a high level view of how the components of an online prediction service communicate. Notice that each component 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 Fig. 5a.

7.2 Rejected Implementation Approaches

We considered four different approaches to implementing communication: CORBA RMI [40], the CORBA event channel service [40, pp. 196-204], Java RMI [44], and SOAP/XML-RPC. We rejected CORBA because it would have required users to buy, install, and understand a CORBA implementation. We rejected Java RMI because we would have had to make extensive use of the Java Native Interface (JNI), and, in our previous experience, JNI is remarkably unportable, especially on uncommon platforms.

The fourth possibility was to use XML-RPC [47] or SOAP [19]. These XML-based approaches are very portable and have been advocated by the Global Grid Forum [45], but the overheads of using an XML representation were judged to be far too high. Later, we implemented XML serialization for an RPS load measurement using a well-regarded XML parser and drew the conclusion that we had made the right decision. Fig. 6 compares RPS's serialization technique with the XML technique. XML requires that much more code be written, the serialization process is much slower, and the serialized objects that would be sent on the network are much larger.

7.3 Implementation

Our mirror implementation, illustrated in Fig. 5b, is a C++ template class that is parameterized at compile-time by handlers for stream input and for request/response input.

Property	RPS Binary Format	RPS XML Format
Size (bytes)	52	851
Pack LOC	10	36
Unpack LOC	10	150
Pack time (μs)	6	287
Unpack time (μs)	16	1500

Fig. 6. Comparing RPS's binary format to an XML format for a host load measurement.

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 to 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.

As shown in Fig. 5b, the heart of a mirror is a select call 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.

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 and perform computation on it, yielding an output data item, which it passes to the mirror's data forwarder.

The data forwarder will then serialize the item to all the open output streams. If a particular output stream is not writable, it will buffer the write and register a handler with the selector to be called when the stream is once again writable. 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, and then register the handler with the connection manager.

The I/O multiplexing approach that the mirror implements ensures ready portability to different platforms, including very simple ones that provide no thread or process support.

The mirror class knows about a wide variety of different transport mechanisms, in particular, TCP, UDP (including multicast IP), Unix domain sockets, and pipes or file-like entities. Endpoints for each of these forms of communication can be created dynamically at runtime and there is no RPS limit on the number of clients that can attach or listen to a stream.

7.4 Special Communication Mechanisms

We have developed two specialized communication mechanisms for use with RPS with the goal of minimizing the amount of network traffic. The first mechanism, wavelet transport, seeks to convey a signal only at the maximum sample rate across all interested consumers. The signal is wavelet transformed at the source, creating multiple detail signals with geometrically decreasing rates, each of which is multicast on a different channel. A consumer subscribes only to those signals necessary to reconstruct the signal at the resolution (and sample rate) it needs. Wavelet transport also serves to decouple the producer's sample rate from that of the consumer.

This scheme, which is described in detail in a previous paper [42] and included in the RPS implementation, unfortunately has a problem. The sample delay through the wavelet transform, which is a function of the wavelet type and the number of levels in the transform, coupled with a low sample rate, leads to a high real-time delay in receiving information. There is some hope that this problem can be addressed, but we have not yet solved it.

We have also built RPS communication on top of our Diffusion system [10], [8] on Linux. The Diffusion kernel module allows us to piggyback additional data on existing packet transfers, exploiting unused header fields and trailer padding to carry the data. Although this communication method provides no guarantees, it allows us to absolutely minimize RPS's communication traffic, down to zero. One scenario in which it is quite effective is communicating host load information on an Ethernet LAN, in which case, the data is piggybacked on the frequent ARP broadcasts made by each host.

8 Components

A user of RPS can choose to straightforwardly incorporate elements from the sensor library (Section 4), the time series library (Section 5), the wavelet library (Section 6), and the communication library (Section 7) in any C++ program. However, the easiest way to use RPS is to compose the prepackaged components that are provided.

Each component is a program that implements a specific RPS function. 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 runtime reconfiguration and to permit multiple applications to use their services. In Section 9.1, we compose an online host load prediction system out of the components we describe here.

RPS includes over 40 components that fit into five basic groups: sensors, measurement converters, buffers, prediction components, and wavelet components.

Sensor components encapsulate the sensors of Section 4, and generate streams of sensor-specific measurements. Related components provide mechanisms to control the sensors and read the measurement streams.

The remainder of RPS operates on a generic measurement type. Hence, each type of sensor is paired with a measurement converter. Also, a special converter is provided that can simply read a stream of numbers from standard input, making it straightforward to couple non-RPS sensors with the system.

For the most part, communication and computation in RPS-based systems is event-driven. A sensor generates a stream of measurements that drive the other components. When a component receives a measurement, it computes a response and pushes it to its downstream components. However, RPS-based systems also often need memory. For example, a predictor component that needs to refit its model needs a history of data up to the present time. In addition, external components (clients) need to be able to query the system asynchronously. Buffers serve both of these purposes. A buffer stores the last n items of a stream of raw sensor data, generic measurement data, predictions, or wavelet coefficients, and lets other components request the last $k \leq n$ of them at any time.

Most of the prediction components are event-driven, providing continuous prediction services for generic measurement streams. The main component, when started, retrieves a measurement sequence from a buffer, 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. The actual work is done by a subprocess, limiting the impact of a crash caused by a bad model fit.

The core prediction component 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. The parameters can be chosen by the user, through a client, or, alternatively, an evaluation component can be run that continuously evaluates the quality of the predictions and forces a model refit when prediction quality exceeds limits set by the user.

Prediction components that provide stateless, request/response prediction services are also provided. Here, a client can send a sequence of measurements and a model to a server which fits the model to the sequence creates a predictor and returns predictions for the next m values of the sequence.

Similar to prediction components, wavelet components are available both for use in a streaming framework and a request/response framework. In both cases, forward and reverse transforms of the types described in Section 7 are possible.

It is important to note that the set 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.

A client component, one that reads from a stream or a buffer or that reconfigures another component can be implemented in as few as eight lines of RPS-specifc C++ code, while a bare server component, including both stream and request/response functionality takes as few as 12.

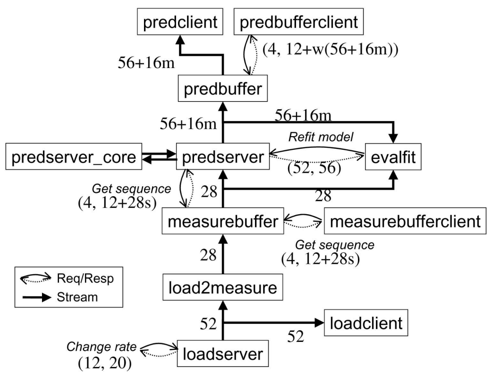

Fig. 7. Online host load prediction system composed out of the RPS components described in Section 8.

9 Runtime Costs

The RPS-based components described in the previous section are composed at startup time to form online prediction systems. To evaluate the performance of RPS for constructing such systems, we constructed an RPS-based prediction system for host load 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.

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 rates are two to three 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 the design, the actual delivered performance depends on the particular composition of components, the communication mechanisms employed, and the predictive models being used. Recall that the computational costs of RPS's models cover several orders of magnitude.

9.1 Host Load Prediction System

Our host load prediction system implements the functionality of Fig. 1 using the RPS components. It uses a predictive model, an AR(16), that we have found appropriate for host load prediction in other work [14] and have used to provide both predictions of the running time of tasks [11] and soft real-time scheduling in a distributed environment [12].

Fig. 7 shows the configuration of the host load prediction system. The boxes in the figure represent components, while the dark arrows represent stream communication between components and the symmetric arrows represent request/response communication between components. The arrows are annotated with communication volumes, in bytes, 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 measurement buffer while m is the number of steps ahead for which predictions are made and w is the number of predictions being requested from the prediction buffer.

The system works as follows: The loadserver component periodically measures the load on the host on which it is running. Each new measurement is forwarded to any attached clients 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. The model is an AR(16) fit to a 600-sample window and providing predictions for 1 to 30 steps into the future.

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 components is an interesting one because it is very conservative. By running all the components on a single host, we maximize the impact on the host and minimize the possible rate. Using TCP communication instead of local mechanisms like Unix domain sockets or pipes further increases the impact. If RPS can achieve reasonable performance and low overhead levels in such a suboptimal configuration, it is likely the case that a performance-optimized RPS-based system would do at least as well. We discuss such a system later.

9.2 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 5 that the host load sensor library requires only about 1.6 μs to acquire a sample. As for the cost of prediction, Fig. 4 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 Fig. 7 indicates that, for 30-step-ahead ($m = 30$) predictions, eight messages are sent for each cycle. There are three 28-byte messages, two 52-byte messages, and three 536-byte messages. The measured bandwidths of the host for messages of this size are 2.4 Mbytes/s (28 bytes), 4.2 Mbytes/s (52 bytes), and 15.1 Mbytes/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 the system's rate of about 3.1 KHz.

The host we evaluated the system on has a timer interrupt rate of 1,024 Hz, which means that 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.

9.3 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 1,024 Hz in powers of 2.

9.3.1 Prediction Latency

In an online prediction system, the timeliness of the predictions is paramount. 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 Fig. 7.

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. Fig. 8 shows that this is indeed the

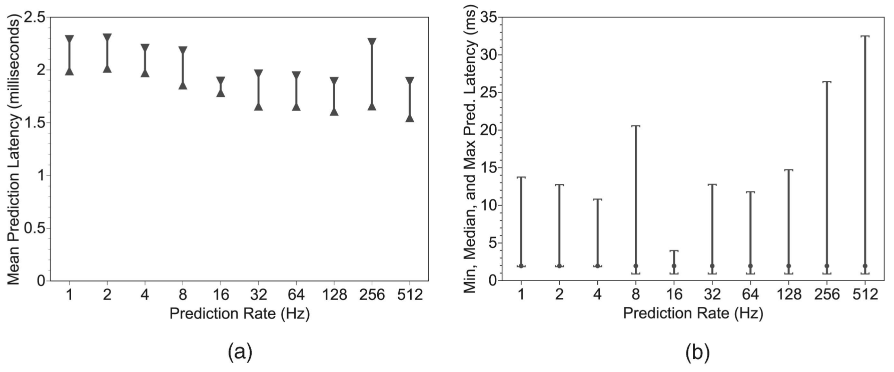

Fig. 8. Prediction latency as a function of measurement rate: (a) 95 percent confidence interval of mean latency and (b) minimum, median, and maximum latency.

case. Fig. 8a plots the 95 percent confidence interval for the mean prediction latency as a function of increasing measurement rates. We do not plot the latency for the 1,024 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.

Fig. 8b plots the minimum, median, and maximum prediction latencies as a function of increasing measurement rate. Once again, we have elided the 1,024 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.

9.3.2 Resource Usage

To measure the CPU usage of our representative host load prediction system, we did the following: First, we started an independent sensor measuring the CPU utilization. We then started the prediction system at its default rate of 1 Hz and let it quiesce. Next, we started a client to receive the prediction stream and let the system quiesce. Then, we swept the measurement rate from 1 Hz to 1,024 Hz in powers of 2. For each of the 10 rates, we let the system quiesce. Finally, we reset the rate to 1 Hz. Fig. 9 shows plots of what the sensors recorded over time. Fig. 9a shows the percentage of the CPU that was in use over time, while Fig. 9b breaks down the system and user components. The system component is essentially the time spent doing TCP-based IPC between the different components.

There are several important things to notice about Fig. 9. First, we can sustain a measurement rate of between 512 Hz and 1,024 Hz on this machine. The exact rate is 720 Hz. While this is nowhere near the upper bound of 3.1 KHz that we arrived at in Section 9.2, it 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).

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 percent, while, at 16 Hz (the closest rate to 14 Hz), it is about 5 percent. For comparison, the "background" CPU usage measured when only running the sensor is itself around 1.5 percent.

The bandwidth requirement of the system is $1,796 \times f_s$ bytes/second, where f_s is the measurement rate. 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 percent 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,

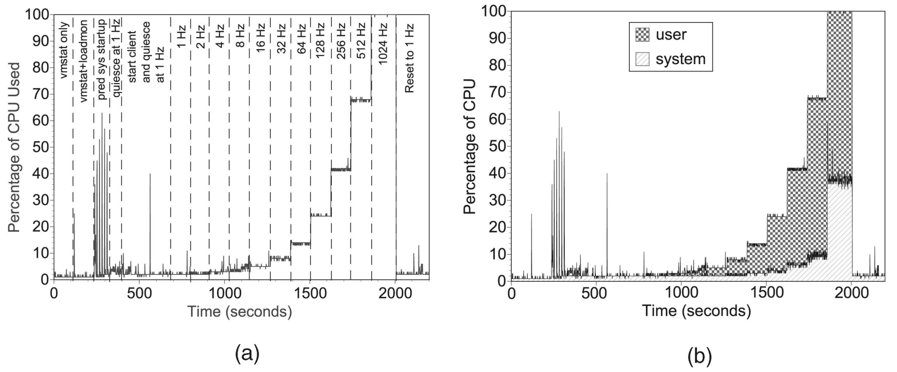

Fig. 9. CPU utilization produced by system. The measurement rate is swept from 1 Hz to 1,024 Hz. (a) Total percentage of CPU used over time. (b) A system and user time breakdown.

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 %

Fig. 10. Maximum measurement rates achieved by monolithic and composed host load prediction systems.

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, 1,900 hosts could operate in 1 MB/s.

9.3.3 Monolithic System

The composed host load prediction system we have described so far can operate at a rate 52–720 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 percent of the upper bound we determined in Section 9.2. To determine if higher rates are indeed possible, we implemented a monolithic, single process host load prediction system using the RPS libraries directly.

Fig. 10 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, and the percentage of optimal rate that each configuration achieves. The message here is two-fold: First, for typical resource measurement rates, even a simple system, composed at runtime with no programming, has an overhead that is low enough to be very practical. Second, it is possible to use RPS at the level of libraries to build a system that comes near to the optimal possible overhead.

10 Prediction Accuracy

A natural question to ask about a prediction system is how good its predictions are. Unfortunately, it is an ill-defined question because prediction accuracy depends on

1. the nature of the signal—its inherent predictability, which is effectively unmeasurable,
2. the sampling of the signal,
3. the models that are available, and
4. whether an appropriate model is chosen. Furthermore,
5. the practical aspects of the model, such as its fit and step costs, are critical.

RPS addresses points 3 and 5 by providing a many models with efficient implementations.

Addressing 1, 2, and 4 necessarily combines automation and human input. We have shown how to use RPS's wavelet analysis to efficiently *decouple* the sample rate from queries an application makes, but it is still necessary for the RPS user to determine an appropriate rate for the resource and then implement a non-aliasing sampler. RPS addresses (4) by providing automatic model refitting, but a human must choose the underlying model. Of course, multiple-expert techniques [5] in which we run many predictors simultaneously can be employed here as in NWS. With complex predictors, however, the cost of doing this grows very quickly.

Using RPS, we have developed prediction systems for host load [14], network bandwidth [37], and various Windows performance data [24], [25]. We summarize the host load and network bandwidth results here.

Host load, sampled at a 1 Hz rate, is readily predictable from 1 to 30 seconds into the future using AR(16) models (the system of the previous section) [14]. The predictions of values and error are accurate enough to compute tight confidence intervals for the running time of short tasks [11]. These predictions are sufficient to do effective predictive scheduling of distributed soft real-time tasks [12].

We explored passive network bandwidth prediction using multiscale techniques, including binning (Haar wavelets) and higher order wavelet analysis, looking at timescales from 1 ms to 1,024 s [37]. There is no clear statement to be made about the predictability of network traffic in general. In situations where it is predictable, high-order threshold autoregressive models are most appropriate. If predictable, the traffic tends to be very predictable, even with very simple models. It is not the case that prediction monotonically improves with timescale. Given these observations, we have moved in the direction of active measurement and prediction [31], [30].

11 Retrospective

RPS has gone through three major versions, including the addition of many predictive models, the entire wavelet toolkit, and different communication schemes, all while keeping the same basic design. Overall, its flexibility, extensibility, and reasonable performance have been proven. There are, however, a number of things that we would change if we were to design a new resource prediction toolkit from scratch.

A weakness of RPS is that it is based on univariate signals. This design decision has implications throughout most of the system. It would have been better to base RPS on a multivariate signal model. Cross correlation between signals is a real and exploitable feature for prediction.

RPS components are designed only to be composed at startup time or runtime. This makes creating new prediction systems trivial, but, to build a monolithic system, the programmer has to operate at the level of RPS's libraries. While this is not difficult, it would be better if the programmer could generate both composed and monolithic systems from a common description.

RPS communicates metadata in-band—every serializable RPS data structure includes the necessary metadata to make sense of it in isolation. The communication costs would be even lower if metadata were communicated out-of-band, perhaps through a directory service.

RPS's low-level communication mechanisms and core implementation language (C++) have helped make it easy to port to new platforms. However, because of the evolving nature of C++, porting to newer versions of a particular *compiler* has proven to be time-consuming.

RPS contains no naming or directory services. The user is responsible for finding the elements of a prediction system. This is eased somewhat through canonicalization of locations in the various scripts provided with RPS. While this can be painful at times, we argue that it is outside of the purview of a toolkit and that, for deployed prediction systems, directory services should be responsible for this sort of information.

12 Conclusion

We have designed, implemented, and evaluated RPS, a toolkit for constructing online and offline resource prediction systems in which resources are represented by independent, periodically sampled, scalar-valued measurement streams. RPS consists of resource sensors, an extensive time series library, a sophisticated communication library, an extensive wavelet analysis library, and a set of component programs out of which resource prediction systems can be readily composed at run time. The performance of RPS is quite good in terms of the maximum sample rates supported, the prediction latency, and the bandwidth requirements. These results demonstrate the feasibility of resource prediction in general and of using RPS-based systems for resource prediction in particular. RPS can be acquired from http://rps.cs.northwestern.edu.

Acknowledgments

A number of people have contributed to the implementation of parts of RPS, including Jason Skicewicz (wavelet library), Mike Knop (Microsoft Windows sensor), Luka Spoljaric (Linux /proc sensor), and Alex Shoykhet (Web interface). Extensive discussions with David O'Hallaron, Bruce Lowekamp, Thomas Gross, and Jaspal Subhlok were immensely helpful. This effort was sponsored by the US National Science Foundation under Grants ANI-0093221, ACI-0112891, ANI-0301108, EIA-0130869, and EIA-0224449.

References

[1] M. Aeschlimann, P. Dinda, L. Kallivokas, J. Lopez, B. Lowekamp, and D. O'Hallaron, "Preliminary Report on the Design of a Framework for Distributed Visualization," *Proc. Int'l Conf. Parallel and Distributed Processing Techniques and Applications (PDPTA '99)*, pp. 1833-1839, June 1999.

[2] S. Basu, A. Mukherjee, and S. Klivansky, "Time Series Models for Internet Traffic," Technical Report GIT-CC-95-27, College of Computing, Georgia Inst. of Technology, Feb. 1995.

[3] J. Beran, "Statistical Methods for Data with Long-Range Dependence," *Statistical Science*, vol. 7, no. 4, pp. 404-427, 1992.

[4] F. Berman and R. Wolski, "Scheduling from the Perspective of the Application" *Proc. Fifth Symp. High Performance Distributed Computing*, pp. 100-111, Aug. 1996.

[5] A. Blum and C. Burch, "On-Line Learning and the Metrical Task System Problem," *Proc. 10th Ann. Conf. Computational Learning Theory (COLT '97)*, pp. 45-53, 1997.

[6] G.E.P. Box, G.M. Jenkins, and G. Reinsel, *Time Series Analysis: Forecasting and Control.* Prentice Hall, 1994.

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

[8] B. Cornell, J. Lange, and P. Dinda, "An Implementation of Diffusion in the Linux Kernel," Technical Report NWU-CS-02-12, Dept. of Computer Science, Northwestern Univ., Sept. 2002.

[9] P.A. Dinda, "The Statistical Properties of Host Load," *Scientific Programming*, vol. 7, nos. 3-4, 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] P.A. Dinda, "Exploiting Packet Header Redundancy for Zero Cost Dissemination of Dynamic resource information," *Proc. Sixth Workshop Languages, Compilers, and Run-Time Systems for Scalable Computers*, Mar. 2002.

[11] P.A. Dinda, "Online Prediction of the Running Time of Tasks," *Cluster Computing*, vol. 5, no. 3, 2002. Earlier version in *HPDC 2001;* summary in *SIGMETRICS 2001.*

[12] P.A. Dinda, "A Prediction-Based Real-Time Scheduling Advisor," *Proc. 16th Int'l Parallel and Distributed Processing Symposium (IPDPS 2002)*, Apr. 2002.

[13] P.A. Dinda and D.R. O'Hallaron, "An Extensible Toolkit for Resource Prediction in Distributed Systems," Technical Report CMU-CS-99-138, School of Computer Science, Carnegie Mellon Univ., July 1999.

[14] P.A. Dinda and D.R. O'Hallaron, "Host Load Prediction Using Linear Models," *Cluster Computing*, vol. 3, no. 4, 2000. Earlier version in *HPDC 1999.*

[15] C. Fraley, "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.

[16] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam, *PVM: Parallel Virtual Machine.* Cambridge, Mass.: MIT Press, 1994.

[17] C.W.J. Granger and R. Joyeux, "An Introduction to Long-Memory Time Series Models and Fractional Differencing," *J. Time Series Analysis*, vol. 1, no. 1, pp. 15-29, 1980.

[18] N.C. Groschwitz and G.C. Polyzos, "A Time Series Model of Long-Term NSFNET Backbone Traffic," *Proc. IEEE Int'l Conf. Comm. (ICC '94)*, vol. 3, pp. 1400-1404, May 1994.

[19] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. Nielsen, "SOAP Version 1.2 Specification," technical report, World Wide Web Consortium, 2003.

[20] D. Gunter, B. Tierney, K. Jackson, J. Lee, and M. Stoufer, "Dynamic Monitoring of High Performance Distributed Applications," *Proc. 11th IEEE Symp. High Performance Distributed Computing (HPDC)*, 2002.

[21] M. Hailperin, "Load Balancing Using Time Series Analysis for Soft Real-Time Systems with Statistically Periodic Loads," PhD thesis, Stanford Univ., Dec. 1993.

[22] J. Haslett and A.E. Raftery, "Space-Time Modelling with Long-Memory Dependence: Assessing Ireland's Wind Power Resource," *Applied Statistics*, vol. 38, pp. 1-50, 1989.

[23] J.R.M. Hosking, "Fractional Differencing," *Biometrika*, vol. 68, no. 1, pp. 165-176, 1981.

[24] M. Knop, P. Dinda, and J. Schopf, "Windows Performance Monitoring and Data Reduction Using Watchtower," *Proc. Workshop Self-Healing, Adaptive, and Self-Managed Systems (SHAMAN)*, June 2002.

[25] M. Knop, P. Paritosh, P. Dinda, and J. Schopf, "Windows Performance Monitoring and Data Reduction Using Watchtower and Argus" (poster), *Proc. Supercomputing*, Nov. 2001. Extended version appears as Northwestern Univ. Computer Science Dept. Technical Report NWU-CS-01-6.

[26] B. Lowekamp, N. Miller, R. Karrer, T. Gross, and P. Steenkiste, "Design, Implementation, and Evaluation of the Remos Network Monitoring System," *J. Grid Computing*, vol. 1, no. 1, pp. 75-93, 2003.

[27] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok, "A Resource Monitoring System for Network-Aware Applications," *Proc. Seventh IEEE Int'l Symp. High Performance Distributed Computing (HPDC)*, pp. 189-196, July 1998.

[28] B. Lowekamp, D. O'Hallaron, and T. Gross, "Direct Queries for Discovering Network Resource Properties in a Distributed Environment," *Proc. Eighth IEEE Int'l Symp. High Performance Distributed Computing (HPDC99),* pp. 38-46, Aug. 1999.

[29] B. Lowekamp, B. Tierney, L. Cottrell, R. Hughes-Jones, T. Kielmann, and M. Swany, "A Hierarchy of Network Performance Characteristics for Gridapplications and Services," Global Grid Forum recommendation, May 2004.

[30] D. Lu, Y. Qiao, P. Dinda, and F. Bustamante, "Characterizing and Predicting TCP Throughput on the Wide Area Network," *Proc. 25th Int'l Conf. Distributed Computer Systems (ICDS),* June 2005.

[31] D. Lu, Y. Qiao, P. Dinda, and F. Bustamante, "Modeling and Taming Parallel TCP on the Wide Area Network," *Proc. 19th Int'l Parallel and Distributed Processing Symp.,* Apr. 2005.

[32] *S-Plus User's Guide,* MathSoft, Aug. 1997, http://www.mathsoft.com/splus.

[33] *MATLAB System Identification Toolbox User's Guide,* The Mathworks, 1996, http://www.mathworks.com/products/sysid.

[34] *MATLAB User's Guide,* The Mathworks, 1996, http://www.mathworks.com/products/matlab.

[35] K. Obraczka and G. Gheorghiu, "The Performance of a Service for Network-Aware Applications," *Proc. ACM SIGMETRICS Symp. Parallel and Distributed Tools (SPDT '98),* Oct. 1997. Also available as USC CS Technical Report 97-660.

[36] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in Fortran.* Cambridge Univ. Press, 1986.

[37] Y. Qiao, J. Skicewicz, and P. Dinda, "An Empirical Study of the Multiscale Predictability of Network Traffic," *Proc. 13th IEEE Int'l Symp. High Performance Distributed Computing,* June 2004.

[38] M. Samadani and E. Kalthofen, "On Distributed Scheduling Using Load Prediction from Past Information," abstracts published in *Proc. 14th Annual ACM Symp. Principles of Distributed Computing (PODC '95),* p. 261, and *Proc. Third Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers,* pp. 317-320, 1996.

[39] S. Seshan, M. Stemm, and R.H. Katz, "Shared Passive Network Performance Discovery," *Proc. USENIX Symp. Internet Technologies and System (USITS),* 1997.

[40] J. Siegal, *CORBA Fundamentals and Programming.* John Wiley and Sons, 1996.

[41] B. Siegell and P. Steenkiste, "Automatic Generation of Parallel Programs with Dynamic Load Balancing," *Proc. 3rd Int'l Symp. High-Performance Distributed Computing,* pp. 166-175, Aug. 1994.

[42] J. Skicewicz, P. Dinda, and J. Schopf, "Multi-Resolution Resource Behavior Queries Using Wavelets," *Proc. 10th IEEE Symp. High-Performance Distributed Computing (HPDC 2001),* pp. 395-405, Aug. 2001.

[43] J.A. Skicewicz and P.A. Dinda, "Tsunami: A Wavelet Toolkit for Distributed Systems," Technical Report NWU-CS-03-18, Dept. of Computer Science, Northwestern Univ., Nov. 2003.

[44] Java Remote Method Invocation Specification, Sun Microsystems, 1997, http://java.sun.com.

[45] B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, and R. Wolski, "A Grid Monitoring Architecture," technical report informational draft, Global Grid Forum, 2002.

[46] H. Tong, *Threshold Models in Non-Linear Time Series Analysis.* Springer-Verlag, 1983.

[47] D. Winer, "XML-RPC specification," technical report, 1999.

[48] R. Wolski, "Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service," *Proc. Sixth High-Performance Distributed Computing Conf. (HPDC '97),* pp. 316-325, Aug. 1997.

[49] R. Wolski, N. Spring, and J. Hayes, "Predicting the CPU Availability of Time-Shared Unix Systems," *Proc. Eighth IEEE Symp. High Performance Distributed Computing (HPDC '99),* pp. 105-112, Aug. 1999.

[50] R. Wolski, N.T. Spring, and J. Hayes, "The Network Weather Service: A Distributed Resource Performance Forecasting System," *J. Future Generation Computing Systems,* vol.15, nos. 5-6, 1999.

[51] J.A. Zinky, D.E. Bakken, and R.E. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems,* vol. 3, no. 1, pp.55-73, Apr. 1997.

Peter A. Dinda received the BS degree in electrical and computer engineering from the University of Wisconsin and the PhD degree in computer science from Carnegie Mellon University. He is an assistant professor of electrical engineering and computer science at Northwestern University with an affiliation with the Northwestern Institute on Complex Systems. His research centers on the intersection of interactive applications and high-performance computing—and, in particular, on frameworks, resource discovery, and online performance analysis and prediction for such applications. He holds the Lisa Wissner-Slivka and Benjamin Slivka Junior Chair of Computer Science at Northwestern. He is a member of the IEEE and the IEEE Computer Society.