Pulsar: A resource-control architecture for time-critical service-oriented applications

M. Astley
S. Bhola
M. J. Ward
K. Shagin
H. Paz
G. Gershinsky

The complexity of real-time systems is growing extremely rapidly, as they move from isolated devices to multilevel networked systems. Traditional methodologies for developing and managing these systems are not scaling to meet the requirements of a new generation of distributed applications. While developers of complex real-time applications are looking to service-oriented architecture to address their needs for ease of development and flexibility of integration, current software infrastructures for service-oriented applications do not address the issue of predictable latency for the applications they host. In this paper, we present Pulsar, a resource-control architecture for managing the end-to-end latency of a set of distributed, time-critical applications. The primary entity of Pulsar is called a controller, which regulates an aspect of resource allocation or scheduling policy. Controllers utilize policy configurations, which may include latency targets to be achieved or resource allocations to be honored, and interact with resource allocators and schedulers (e.g., thread schedulers, memory allocators, or bandwidth reservation mechanisms) to effect local policy. Controllers also provide feedback on how well they are executing a policy. Pulsar includes an application model which captures resource-sensitive behavior and requirements and is independent of high-level programming models and application programming interfaces.

INTRODUCTION

As real-time systems become increasingly complex and accommodate a new generation of distributed applications, traditional methodologies for developing and managing these systems are not scaling to meet the requirements of these applications. While service-oriented architecture (SOA) offers an approach to addressing the needs of developers of complex real-time applications with respect to ease of development and flexibility of integration, the issue of predictable latency for these SOA applications is inadequately addressed by current software infrastructures.

Real-time applications which are deployed over such infrastructures have several key features:

- *Distributed resources*. Components of an application may be distributed, and may compete with other distributed applications for resources such as CPU, networking bandwidth, and memory;
- *Mixed requirements*. The set of applications may include *hard* real-time (i.e., those in which deadlines must be met and all events must be handled) and *soft* real-time applications, as well as non-real-time workloads;
- *Dynamic loads*. Workloads may be event-driven and can vary significantly over time; and
- *Dynamic resources*. The set of available resources (both on the server and the network) may change over time due to failures or other reasons.

For example, a trade execution application for a stock exchange may have components which are distributed (for scalability and fault-tolerance) over a cluster of servers. The application is responsible for providing timely data from the market, as well as accepting and executing orders to buy or sell securities. The load experienced by the application is heavily influenced by trade volume, which varies both predictably (e.g., due to the trade backlog at market open) and unpredictably (e.g., due to breaking news). In addition, the application may service a variety of clients with different requirements and importance. Large investment banks, for example, are willing to pay a high premium to ensure a bounded latency on trade execution. Individual investors, on the other hand, may settle for a low-cost, best-effort service.

Satisfying the timeliness requirements for such an application is a challenging problem. For example, the dynamic load and availability of the system prohibit traditional techniques such as static allocation and scheduling. Instead, the system must be able to shift resource allocation spontaneously as needed. Likewise, the system must optimize resource allocation according to differing service level and latency requirements. For example, clients may express their requirements as a service level agreement (SLA), which quantifies the importance and flexibility of meeting requirements at different service levels. SLAs can be used to allocate resources in a manner which optimizes overall benefit to clients. Finally, scalable and fault-tolerant control mechanisms are needed to minimize overhead and provide resiliency in the face of server and network failures.

In this paper, we present Pulsar, a resource-control architecture for managing the end-to-end latency of a set of distributed, time-critical applications. Pulsar applications are described using a programming model that captures resource-sensitive behavior and requirements that are independent of high-level programming models and application programming interfaces (APIs). In particular, timeliness requirements are modeled as utility functions which map utility as a function of end-to-end latency. Depending on the shape of the utility function, the system is able to make trade-offs between total utility and available resources. These trade-offs are reflected in resource-control policies which are distributed among the nodes of the system.

Resource-control policies are constructed and enforced using controllers, which are arranged in a hierarchy in order to regulate various aspects of resource allocation or scheduling policy. Top-level controllers establish overall resource-control policies based on utility trade-offs. These policies are passed to "child" controllers and may include latency targets which must be achieved or resource allocations which must be honored. Intermediate controllers (e.g., one per server) translate policies received from parent controllers into policies delegated to child controllers. At the lowest level, controllers interact directly with resource allocators and schedulers, e.g., thread schedulers, memory allocators, or bandwidth reservation mechanisms, in order to effect local policies.

In this paper, we present: (1) a distributed, hierarchical control mechanism for distributed real-time applications; (2) an abstract model of resources which allows high-level control of these applications without exposing explicit resource-control mechanisms (e.g., resource pooling and sharing); (3) a framework for utility-based optimization which maximizes total system utility based on a novel intermediate model of application requirements; and (4) a method for online feedback and error correction to improve system performance.

The paper is structured as follows. We present a programming model for real-time applications in the next section. This is followed by a description of an architecture which supports this model. We dem-

onstrate our approach by way of a distributed power-line fault detection application which we describe in the section "Scenario: Power-line fault detection." In the section "Evaluation," we evaluate the performance of our architecture in this scenario. We review related work in the subsequent section, followed by our conclusion.

PROGRAMMING MODEL

Whereas real-time applications typically target a specific high-level programming model, such as the Real-time CORBA** specification[1] or the Real-time Specification for Java**,[2] our techniques are focused on low-level resource management mechanisms which are often independent of high-level APIs. As a result, in this paper we describe applications in terms of a low-level model which captures resource-sensitive behavior and requirements. We view our model as a possible intermediate target for high-level APIs and languages, including those that facilitate building service-oriented applications like Web Services Business Process Execution Language (WS-BPEL)[3] and Service Component Architecture (SCA).[4]

Applications with real-time deadlines are defined in terms of jobs, job sets, and job flows. A *job* is a discrete unit of computation that is localized on a node in the system. Jobs may execute concurrently, and all jobs are preemptible. Constraints among jobs (e.g., mutual exclusion) may be specified in job sets as described below. Jobs are organized into *job sets* consisting of a job list (a list of jobs contained in the set), a dependency graph (a directed acyclic graph [DAG] with a unique root, one or more leaf nodes, vertices which represent jobs, and directed edges which represent dependencies between jobs), and trigger events (a set of external events which cause the job set to be executed).

The unique root of a job set is called the *start job* and the leaf nodes are called *end jobs*. An edge in a job set dependency graph is either a local edge or a remote edge, according to whether the linked jobs execute on the same node or different nodes. Dependencies between jobs indicate ordering constraints, conditional paths, mutual exclusion, or communication. We view job sets as types, instances of which are created when the system receives an appropriate triggering event and agrees to schedule a start job for execution. Due to conditional paths, the jobs executed for a given instance of a job set may differ from other instances of the same job set.

In terms of execution, a job set instance defines a simple data flow which starts when the start job receives a trigger event. Edges carry discrete information in the form of events, and a job in a job set is eligible for dispatch when all inbound edges contain an event. When a job completes execution, it may generate an event on each outgoing edge. Events are used for modeling purposes and may not always have a physical realization. For example, the implementation of edges used to indicate ordering, conditional execution, or mutual exclusion need not transmit actual events. Communication edges, in contrast, transmit events representing data.

A *job flow* is a sequence of job set instances with the same timeliness requirements (e.g., a specific deadline for each job set). At run time, each job flow receives a stream of trigger events which cause the appropriate job set instances to be instantiated and scheduled for execution.

Figure 1 shows an example of the application model. In the figure, we illustrate the history of a particular job flow (F1) which shows the release time and execution time of multiple job set instantiations (two instantiations of job set type S1 and one of S2.) The instantiations of S1 differ in their job execution paths (e.g., due to conditional execution.) S2 has the same timeliness requirements as S1 and is included in the same flow. The view shown in the figure is a history because the job set rendering shows the actual jobs executed, not all possible paths. The latter may be different due to conditional execution. For clarity, we have shown job set executions which are non-overlapping in time, though there is no such restriction in the model.

This application model embodies the main features of service orientation. Jobs naturally map to services and are not tied to any implementation technology. Communication between jobs in a job set may be through procedure invocation, either local or remote, or by message passing. The model supports both request-response and one-way message exchange patterns; one-way message exchange may be one-to-one or one-to-many (we are investigating extensions to support many-to-one message exchange, if this is necessary.) We have validated this model with the WS-BPEL and SCA standards, two

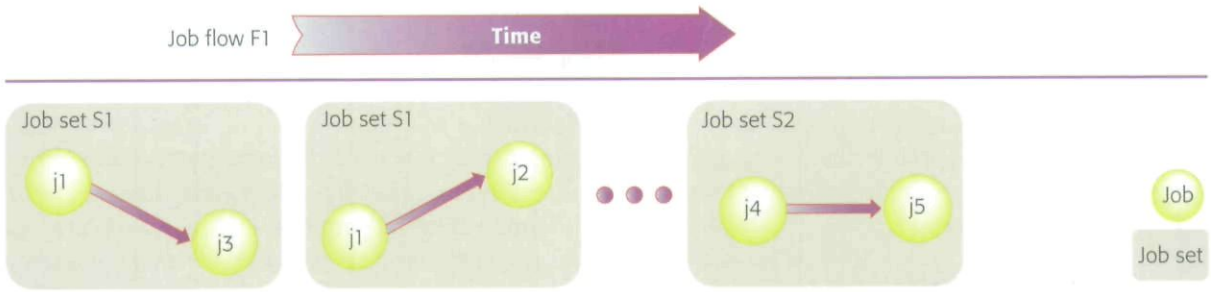

Figure 1
Application model example

leading methods for specifying service-oriented applications. The Pulsar model fully supports the SCA Assembly Model for aggregating components and linking them through wiring. WS-BPEL specifies a rich set of structured activities for prescribing the order in which a collection of activities is executed. The current Pulsar model does not support the <while> and <repeatUntil> structured activities; we are investigating ways to support restricted cycles in the dependency graph if time-critical applications present such requirements.

Timeliness requirements

Application timeliness requirements are specified using a utility function, which is a non-increasing function that maps job set latency to a utility value. The maximum allowable latency may be limited by a critical time beyond which latency may not extend, regardless of utility. Thus, critical time is analogous to a deadline.

Utility functions are a generalization of simple deadlines where, in addition to defining absolute constraints (e.g., latency must not exceed the critical time), the shape of the function can be used to derive trade-offs between latency (i.e., resource allocation) and benefit to the application. Our goal is to satisfy all application deadlines (i.e., critical times) while maximizing utility.

The latency (and hence utility) of a job set depends on the latency experienced by individual jobs, which further depend on resource allocation and may vary according to application parameters. Job flows are expected to define properties which help to determine the latency for jobs (e.g., worst-case or average-case execution time), as well as other resource requirements (e.g., network bandwidth). Such properties could be derived from or corrected by runtime measurements. We can combine these specifications (including trigger-event specifications) with a model of resources to derive the predicted latency for a job.

In the case of soft real-time flows, it is often inefficient to specify performance as a function of worst-case execution since worst-case latency is often much larger than average latency. Our programming model allows flow latency to be based on particular latency samples (e.g., the 95th percentile). Likewise, our implementation uses these samples for monitoring and fine-tuning resource allocations.

Analysis

The analysis of job flow and resource allocation combines models of job flow latency and resource utilization to maximize a system scheduling objective. In our system, the scheduling objective is to maximize the sum of the utilities of the job flows, while ensuring that the latency of all job set instances remains below the critical time for the job flow it belongs to.

To perform scheduling analysis, the following information must be extracted from the application model: the timeliness requirements (i.e., utility function) and dependency graph for each job set; the inter-release time of the start jobs for job set instances in each flow; the probability of observing particular dependency graph instances for each job set (this implies knowing the probability of following each conditional branch in the dependency graph); and the execution time (e.g., worst-case execution time) for each job.

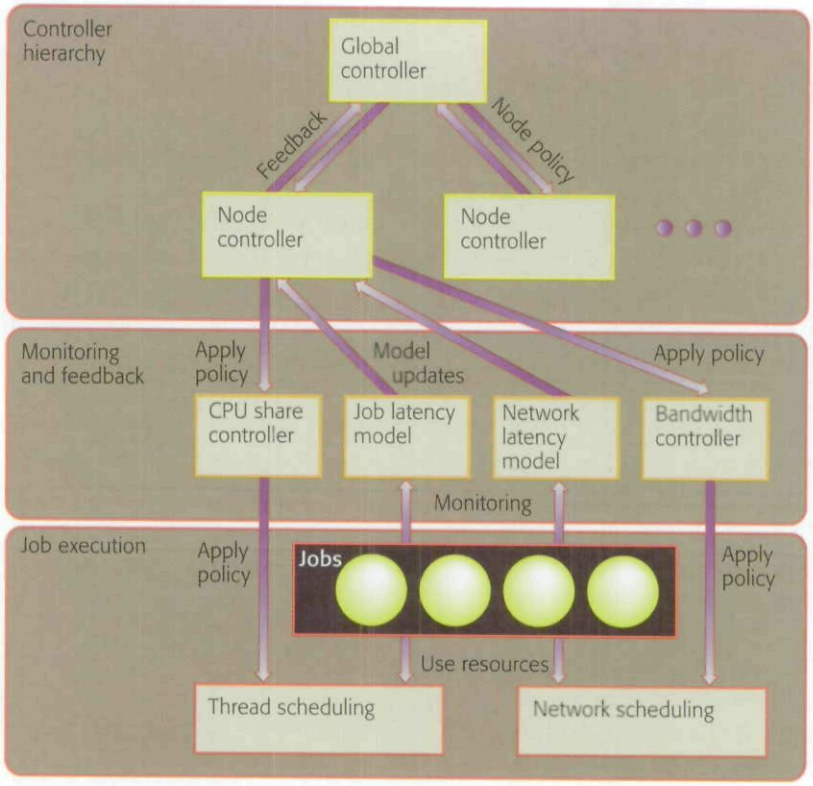

Figure 2
High-level view of Pulsar system architecture

It may not be possible to extract such information from all applications. In extreme cases, it may be necessary to estimate application behavior from an online model. Our system performs a limited form of such estimation by way of error correction. However, more-complete online modeling is being considered for future work. Likewise, we consider only deterministic job dependency graphs in this paper; graphs with probabilistic branches are being considered for future work.

A latency model for a flow can be constructed by considering the latency of each job set. The latency of a job set, in turn, can be modeled according to the latencies due to jobs and edges. Note that both kinds of latencies include queuing delays of various kinds (communication buffer queues, thread scheduler queues, etc.). However, latency at a non-communication local edge is assumed to be zero.

The latencies of jobs and edges are affected by resource allocations, such as CPU share, bandwidth, and memory. In our approach, we define node-specific resource models which describe the effect of a particular allocation on latency. For example, increasing CPU share for a particular job will decrease its latency but may increase latency for concurrent jobs, since less of this resource is available.

Depending on how the latency models are constructed, it may be possible to perform a cost-versus-benefit analysis which leads to an optimal allocation strategy. We illustrate this technique using a worst-case latency model in the section "Evaluation."

ARCHITECTURE AND IMPLEMENTATION

Figure 2 presents a high-level view of the Pulsar system architecture. This architecture consists of three main components: the controller hierarchy, job execution, and monitoring and feedback. The controller hierarchy determines optimal resource allocations from node-level models of resource utilization. The job execution layer executes jobs and updates local models to correct for modeling error or other performance differences. We describe each of these components in more detail in the following.

Controller hierarchy

The controller hierarchy subdivides the task of global resource allocation between a node controller (one for each node), which utilizes specific models of resource requirements, and a global controller, which combines resource models with latency and flow utility models to determine appropriate resource allocations (see the section "Programming model"). The global controller does not need to be centralized. For example, a global controller may be created per flow with each controller interacting with the others to derive resource allocations.

Global controllers determine resource policies (i.e., specific resource assignments for each node) and forward these policies to each of the node controllers. Conversely, node controllers use local monitoring to improve local resource models and periodically forward these models to each global controller. The structure of the models and the mechanisms used by the global controllers to determine allocations may vary widely; this is beyond the scope of this paper. However, we consider an explicit example of this approach in the section "Evaluations."

In the current architecture, the node-level latency models describe latency as a function of particular CPU and network share allocations. That is, we assume some form of a proportional share (PS) scheduling[5,6] mechanism but do not require a particular implementation. These models vary according to the underlying resource and modeling error. For example, speed differences in CPUs or the choice of different share scheduling implementations will yield slightly different models. The node controllers attempt to compensate for modeling error by monitoring performance and incorporating error correction into the models.

Node controllers perform the additional function of enforcing resource allocations according to the policies provided by global controllers. In the current architecture, resource allocations are enforced by directing the kernel to assign shares to job threads and network links (see the section "Implementation details").

Job execution

Job execution is handled by two components. The *Job Execution Environment* (JEE) provides a container for the job flows described by the application model. The *TransFab* component augments the JEE by providing access to network resources when required by job set dependencies. The JEE uses an event dispatch model to implement the facilities required by the application model.

At system start-up, an object is instantiated for each job in each job set which may execute in a job flow. Each job object is associated with an event queue. The job objects encapsulate the application code to be executed for the job, and the event queue stores events which represent dependent interactions in a job set instance. Each job object is uniquely associated with a job set type (i.e., job objects are never shared between job set types).

In the current implementation, each job object is also associated with a dedicated thread having a share which is assigned according to the current policy. The events on the event queue are used to dispatch these threads in a serial fashion. The thread is suspended if the event queue is empty.

At run time, a job set instance is represented by its triggering event, which is queued in the event queue of the appropriate start job object. A job set instance is viewed as "released" when its triggering event has been enqueued. During execution, a job object may produce at most one event on each outgoing link, as determined by the job dependency graph. When a job object completes, each such event is enqueued on the appropriate downstream job object queue. If the downstream object queue resides on a different node, then Transfab is invoked to forward the event to that node.

Monitoring and feedback

The observed latency of job flows may differ from that predicted by our models due to error induced by model abstractions and slight variations in run-time performance. Our architecture attempts to compensate for these errors by monitoring performance at the node level and computing smoothed additive error terms for the latency models.

The monitoring mechanisms and computation of error are heavily dependent on the form of the latency models. For example, if latency models are worst-case models, not all latency samples will be worst-case, and filtering will be necessary so that only representative samples are used to adjust the model. Likewise, error correction depends on where