A Subscription-based Monitoring Model for Distributed Object Systems

Kyungkoo Jun, Ladislau Boloni, Ruibing Hao, and Dan C. Marinescu
{junkk, boloni, hao, dcm}@cs.purdue.edu

Computer Sciences Department,
Purdue University,
W. Lafayette, IN 47906

October 5, 1998

Abstract

Distributed object systems are inherently dynamic. The population of such systems is typically very large, the state of objects changes rapidly, and the interactions among the objects can be rather intricate. In such an environment it is essential to provide mechanisms to obtain reliable information about system structures and state, use that information to make decisions, and be notified when the information changes. In this paper, we discuss a *subscription-based monitoring model* and its applications within the Bond monitoring framework, for the implementation of services like fault-tolerance and data source monitoring. Central to our model is the concept of a *probe*, an object used to extend functionality of other objects dynamically. The *Bond System Monitor* and the *Stock Watcher* are two applications of the Bond monitoring framework.

1. Introduction

Distributed object systems have characteristics uncommon to the traditional centralized or even distributed systems. Distributed object systems are in a constant state of flux as utilization and availability of resources change, computers and networks fail, old objects are removed and new objects are added, and software and hardware on existing systems are updated and modified. It is rarely feasible to rely on standard or default system environments when building objects [1].

Objects need to obtain real-time information about system structures and state, use that information to make decisions, and be notified when information changes. The required information spans from low level network activity, processors utilization, to higher level object attributes such as data field values. The update notification needs various models to meet the diverse requirements. Often, the actions taken by an object are the result of a complex analysis of data collected from multiple sources. The mechanisms to gather information and take appropriate actions support *object adaptability*.

We overview briefly two approaches to monitoring in metacomputing. In Legion [11] a set of "phoenix" daemons construct system configuration information by monitoring system processes and restarting them if possible. The phoenix daemons are arranged in a hierarchy: *root* phoenix monitors *cluster* phoenixes, which are monitoring a cluster of hosts by *leaf* phoenixes on each host. Globus [4] offers an *unreliable fault-detection service*, that supports the implementation of application-specific *failure behaviors* such as termination, ignorance, restart, and replication. The failure of a system component is reported late thus the service is unreliable. The fault detection service is based upon two types of entities, *local monitors* that observe the state of the computer where they run and any registered local processes and *data collectors* receive periodic *heartbeat* messages from the local monitors and identify failed components based on missing heartbeats. The system implements a *fail-stop failure detection* mechanism [9].

Chandy and his group have proposed a *global event announce-listen model* [3] to allow objects to make autonomous decisions based on information about system state. The model is based upon a multicasting communication infrastructure that enables objects to periodically announce their local state information as global events and listen for updates from other objects. Objects construct *soft states* to estimates the current state of other objects based on information received earlier.

In this paper we propose *subscription-based monitoring* as an alternative monitoring model. Objects subscribe to other objects and request to be notified of state changes, then analyze the information gathered through monitoring and take specific actions. We present this monitoring model and show how it can be applied to implement various functions needed to enhance system dependability, by supporting fault-tolerance and monitoring of data sources.

We present this monitoring model in the context of Bond [8] a message-oriented distributed object system currently under development at Purdue University. The paper is organized as follows. Next section provides the basic elements of our model. Section 3 provides an overview of Bond and Section 4 introduces the Bond monitoring framework and Section 5 presents two applications.

2. A Subscription-based Monitoring Model

The following terms are used throughout the paper:

- *Property*: an entity consisting of name and its value. In the object-oriented paradigm, a property is implemented as an object variable. Objects have a set of properties and methods that operate on them. Some properties are declared at compile time and some are generated at run-time.
- *State of an object:* the set of properties of an object.
- *Event*: the change of state of an object.
- *Subject:* an object whose properties are of interest to other objects and it is willing to provide them upon request.
- *Monitor:* an object that collects information about the properties of other objects, analyzes the information gathered through monitoring, and takes actions based upon the result of this analysis. Any object can be a subject, a monitor, or both at the same time. For example, object A is a monitor for B and it is being monitored by C.

Our monitoring model is based upon the following principles:

- Monitors and subjects communicate using point-to-point message passing.
- A monitor subscribes to the properties of interest of a subject. In turn, the subject notifies the subscriber of the property changes.
- Monitors operate based upon an event-action model. The actions are specified in terms of perceived states. For instance, if temperature exceeds 100F, raise an alarm.

An implicit assumption of our model is that monitors and subjects are aware of the effects of communication and processing delays. In other words if we denote by δ the sum of all delays from the instance an event occurs, till the action triggered by the event is completed the objects involved in monitoring maintain their relevant properties stable for at least Δ units of time with $\Delta > \delta$. We call this *Δ-monitoring*. Each pair (monitor, subject) needs to agree upon the value of Δ and consider only actions compatible with this value.

At subscription time, a monitor specifies a *policy* consisting of properties *of interest* and a *notification pattern*. The notification patterns supported by our system are *time-triggered*, *change-triggered*, and *hybrid*. In the time-triggered pattern, the subject notifies periodically the monitor using an agreed-upon time interval. In the change-triggered pattern, any change of properties of interest triggers a notification. The hybrid pattern guarantees that the interval between any two consecutive change-triggered notifications does not exceed a predefined value. It performs like change-triggered when changes do occur or as time-triggered if there are no changes.

A first application of the subscription-based monitoring model outlined above is fault-detection. Consider a system with a set of critical servers. A system monitor ensures that all servers are running at all times. The system monitor subscribes to each server requesting a time-triggered notification. The absence of a notification message triggers the desired action, e.g. restarting the server. In Section 5, we present the *Bond System Monitor* [6].

Monitoring of data generators/sources is another application of the model. Examples of data generators are sensors and programs. A Web server providing real-time stock quotes is a data generator. In this case client applications have a vested interest in specific events that can be inferred from the data provided by the server. In this example client applications are monitors and the stock server is a subject. A client subscribes to the server by providing a list of stocks and a notification pattern. The actions taken by the clients range from a detailed analysis of the market to sending E-mail to a list of customers. In Section 5, we present the implementation of the *StockWatcher* and *StockClient* within the Bond monitoring framework.

3. Bond

Bond is a Message Oriented Middleware, MOM, for network computing. It consists of a distributed object communication fabric, servers, and agents. The servers are active objects providing services whose life time is longer than the a Bond session, e.g. directory service, authentication, persistent storage, versioning. Agents are active objects providing scheduling, brokerage, and so on. More information about Bond is available elsewhere [5], [8].

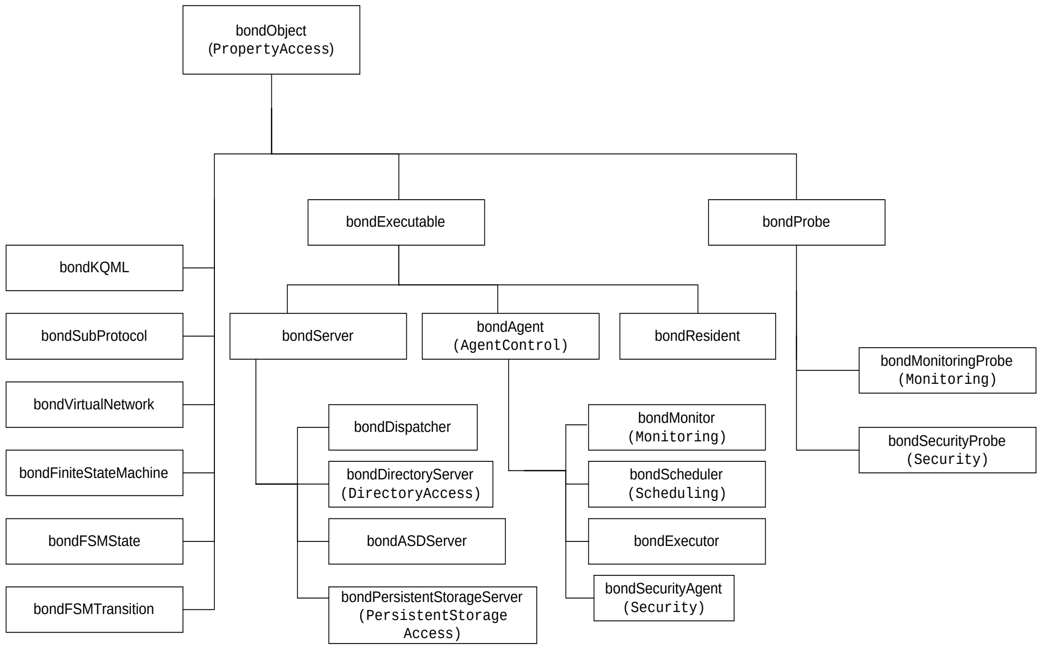

Figure 1: The upper part of the Bond object hierarchy. The sub-protocols implemented in class definitions appear in parenthesis. Every object understands the property access subprotocol implemented in bondObject**.**

Bond objects are persistent network objects, that communicate with each other, can be instantiated and run remotely, and be saved on persistent storage. The upper segment of the Bond object hierarchy is presented in Figure 1. In this hierarchy, *bondObject* implements the common fields of all Bond objects (e.g., name, unique bondID, address, *static fields*, and *dynamic fields*). The static fields are declared during compile time and are implemented as data fields of the class. The dynamic fields are created during run-time and are implemented in the internal hash table.

A property of a Bond object is a pair consisting of a name and a *value object*. The value object can be a single value or a collection of properties. For example, the name could be "CurrentPrice" and the value, the current price of a particular stock.

Bond objects have static and dynamic properties and implement them by means of static and dynamic fields. The following example shows the implementation of a stock quote as a dynamic property. The property consists of name, "IBM Stock" and price and volume properties as value object. Two static fields, *CurrentPrice* and *CurrentVolume* in the class *StockInfo* are for stock price and volume properties. The *get( )* and *set( )* methods implemented in *bondObject* provide access to static and dynamic fields. Both methods check the property names in the static fields before searching the hashtable.

```
Class StockInfo
extends bondObject {
   float CurrentPrice;
```

```
    long CurrentVolume;
}
Class MyProgram
extends bondObject {
    StockInfo Stock = new StockInfo();
    Stock.set("CurrentPrice", 35.66);   //creation of price property
    Stock.set("CurrentVolume", 1400000);//creation of volume property
    this.set("IBM Stock", Stock);       // creation of quote property
    StockInfo IBMStock = this.get("IBM Stock");
}
```

Bond uses KQML, Knowledge Querying and Manipulation Language, [4], as an inter-object communication language. A KQML message consists of a *performative* and a set of order-independent parameters. The performative encodes basic abstractions like asking, replying, achieving, subscribing, etc. Each parameter consists of name beginning with a colon and the corresponding value.

The number and semantics of parameters are dependent on the performative except some *reserved* parameters. The following parameters have the same meaning regardless of performatives: *content* specifies an attitude that the performative expresses, e.g., the name of the called remote function or command; *subprotocol* specifies the subset name which this message belongs to, discussed later; *sender* and *receiver* specify bondIDs.

A *sub-protocol* is a sub-set of KQML messages. It contains messages needed to perform a particular task and is *closed* in the sense that the reply or acknowledgement to a message is always a member of the same sub-protocol. Examples of Bond sub-protocols are listed in [5]. The objects that *implement* a sub-protocol can understand both the syntax and the semantics. For example, two objects that implement *PropertyAccess* sub-protocol can remotely interrogate and set the properties of each other.

The following example shows the KQML message format, left, and an example of a message, right. In this example the performative is *ask-one*, the contents of the message is *get*, the sender and the receiver of the message are identified by their bondIDs, and the sub-protocol is the PropertyAccess. The message requests the receiver to reply with the value of the temperature property.

(performative :content *content*	(ask-one :content get
:parameter1 *value1*	:property temperature
:parameter2 *value2*	:sender *bondID*
:sender *sender*	:receiver *bondID*
:receiver *receiver*	:sub-protocol PropertyAccess)
:sub-protocol name)	

Every incoming message is delivered to the *say( )* function of a receiver. The *say( )* function implements the message parsing and handling. The *say( )* function, is either inherited from a parent or overridden in an object.

The sub-protocol implementation is also inherited. If an incoming message is not understood, it is passed to the *say( )* function of the immediate ancestor in the object hierarchy and so on. Every object can understand the PropertyAccess subprotocol implemented by the root of the object hierarchy (Figure 2).

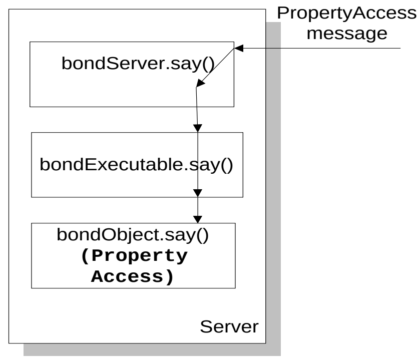

Figure 2: The handling of an incoming message. The message is received by a bondServer object, a descendent of a bondExecutable, which in turn is a descendent of the bondObject. The message is passed along the hierarchy to the *say ( )* function of the bondObject which implements *PropertyAccess* sub-protocol.

Bond uses the concept of *probe* [5] to add new functionality to a Bond object dynamically. A probe is an object that implements a certain sub-protocol and it is attached to an object as a dynamic property. The implementation of a *bondObject* ensures that when an object cannot understand a message by tracing the object hierarchy, it searches the dynamic properties for a probe that can handle the sub-protocol of the message and leaves the message handling to the probe, as shown in Figure 3.

An advantage of the probe is that objects need not implement a particular subprotocol themselves or be within a certain sub-protocol inheritance branch. Bond monitoring framework is based upon the concept of a *probe*, an abstraction that allows us to achieve flexibility and transparency as well as avoid unnecessary overhead. In the following sections we discuss the Bond monitoring framework and the monitoring probe.

4. The Bond Monitoring Framework

To monitor a Bond object means to gather information about its properties. The monitored object is called a *subject* and the monitoring one is a *monitor*. The principle of the Bond monitoring framework is that the monitor creates a local replica of the subject's properties by subscription and takes actions depending on the property values. The subject is responsible for keeping the remote replica of its properties up-to-date.

The Bond monitoring framework is partly based upon the Java event model [10] where the *event listener* registers itself with the *event source* to receive event notifications. However, our framework is designed for distributed object systems while in the Java model events are delivered within a single Java Virtual Machine. Our monitoring framework generalizes the local event model in Java and has a range of applications for distributed object systems such as fault tolerance and data source monitoring as described in Section 2.

The monitoring framework relies on message exchanges between a monitor and a subject that may be running on different hosts. The communication pattern for monitoring is as follows: the monitor sends the subject a *subscription message* containing a list of properties of interest and a notification pattern. The subject replies with a *notification message* containing the requested properties once or repeatedly. The monitor quits the monitoring by an *unsubscription message*. This message pattern is implemented by the *monitoring sub-protocol*, a closed sub-set of KQML messages. The subscription message uses the *subscribe* performative and its parameters. The notification and the unsubscription messages use the *tell* and *discard* performatives with the corresponding parameters. The monitors send the *ask-one* performative when an expected notification message is missing.

Table 1. The monitoring sub-protocol

Performative	:content	Parameters	Description
subscribe	monitor-info	:policy *policy*	ask subject to notify according to *policy*
discard	monitor-stop		ask subject to stop notification
ask-one	monitor-poll		ask subject for immediate notification
tell	monitor-report	:value *value*	notify monitor with *value*

An objective of the monitoring framework is to achieve *object-level transparency*. A monitoring relationship should be established between any pair of objects regardless of monitoring implementations inside the objects. The monitoring functionality should be dynamically attached on demand without affecting the design and behavior of objects. We combine the monitoring model with dynamic attachability of the probe, resulting in *monitoring probe*.

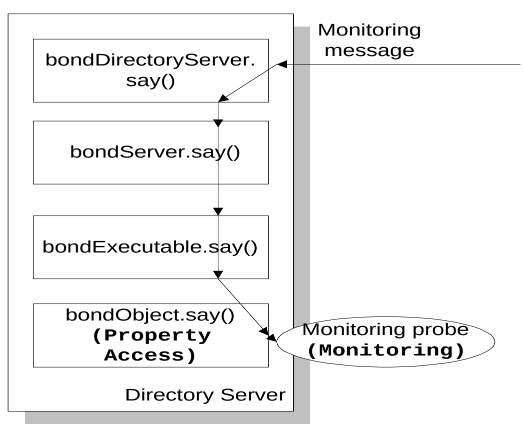

Figure 3: Sub-protocol implementation using probes. The monitoring message is passed along the hierarchy to *say ( )* function of bondObject. After failing to process that message, bondObject searches the dynamic properties of the receiving object, in this case a bondDirectoryServer finds a probe that understands the monitoring sub-protocol and delivers the message to it.

A *monitoring probe* is an object implementing the Bond monitoring model. It understands the monitoring sub-protocol, implements the roles of both the monitor and the subject, and provides a set of APIs. Being a descendent of the probe, a monitoring probe can be attached at run time to an object, as a dynamic property.

An object extended with a monitoring probe understands the monitoring sub-protocol. Figure 3 shows how an object handles monitoring messages. In this example a Bond directory server object is extended with the monitoring probe. An incoming monitoring message is passed along the hierarchy to the *say( )* function of *bondObject*. After failing to understand the monitoring sub-protocol, the *bondObject* searches the dynamic properties looking for a monitoring probe and delivers the message to it.

A monitoring example is depicted in Figure 4. Suppose that there are two objects, A and B, and neither of them understand the monitoring sub-protocol. A is going to monitor B. First, A creates its monitoring probe and provides B's address, the names of the properties of interest, and the desired notification pattern. The monitoring probe encodes the property names and notification pattern and sends it to B. The delivered message propagates along the hierarchy to the monitoring probe created in B. The probe initializes itself, collects the specified properties, and ships them back to A's monitoring probe according to the notification pattern. Notice that B's probe sends messages directly to A's probe avoiding unnecessary message propagation through the hierarchy. A's probe stores the delivered properties as a local replica.

The monitoring framework implemented in the monitoring probe involves three layers: *interface*, *management*, and *sub-probes* layers shown in Figure 5. The interface layer consists of a set of APIs called by objects. The other two layers perform the actual monitoring activities such as notification, un-subscription, etc. The sub-probes layer is a collection of *monitoring sub-probes*. Whenever a monitoring relationship is established, two monitoring subprobes are dynamically created, one at each end, the monitor and the subject. From this moment on all message exchanges for notification and un-subscription, occur between these two sub-probes. At the monitoring termination, both subprobes disappear. The management layer controls the monitoring sub-probes.

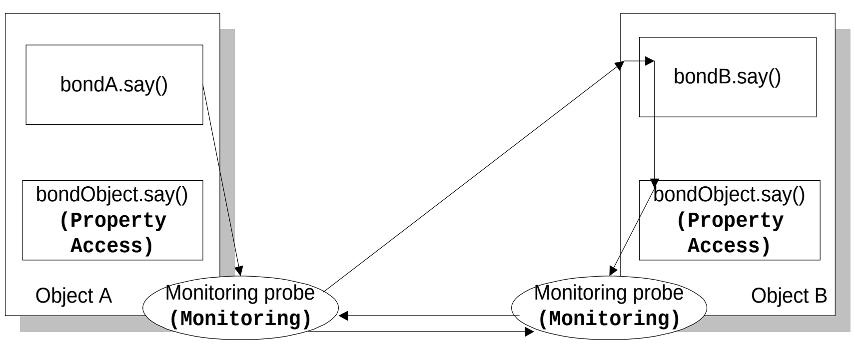

Figure 4: Two objects, A and B, involved in a monitoring relationship, A monitors B. Once the monitoring relationship is set up, all monitoring messages are directly sent to the respective monitoring probes avoiding message overheads.

The interface layer is the only part exposed to outer objects by providing a set of APIs: *startMonitoring( ), stopMonitoring( ), and getMonitors()*. An object can start and stop monitoring using *startMonitoring( )* and *stopMonitoring( )* if the object is supposed to work as monitor. Both methods have a single parameter *policy*, an object containing a subject address, a list of interesting property names, notification pattern, and *manner*. The possible manners for notification are the KQML messages and E-mail. Subjects can obtain a list of the monitors subscribed to it by *getMonitors( )*.

The notification pattern specifies when the subject sends notification messages. It consists of a *model identifier* and three parameters: *start-time*, *duration*, and *interval*. The possible model identifiers are *time-triggered, change-triggered,* and *hybrid.* The semantics of the models were explained in Section 2. The start-time designates the time when the subject starts notification. The duration specifies notification duration. The interval indicates the time between two consecutive notification messages. The time-triggered and hybrid models need all three parameters. For the change-triggered model, only the start-time and duration are needed.

The management layer activates and deactivates the monitoring sub-probes. On the monitor side, the activation and deactivation of the subprobes are triggered by calling *startMonitoring( )* and *stopMonitoring( )*. On the subject side, subscription and un-subscription messages create and destroy the monitoring sub-probes respectively.

An object may monitor other objects while it is being monitored. To support this dual function its monitoring probe contains a set of monitoring sub-probes, some allowing the object to monitor other objects, others allowing it to be the subject of monitoring by other objects. The management layer has two internal table to keep track of running sub-probes according to their roles.

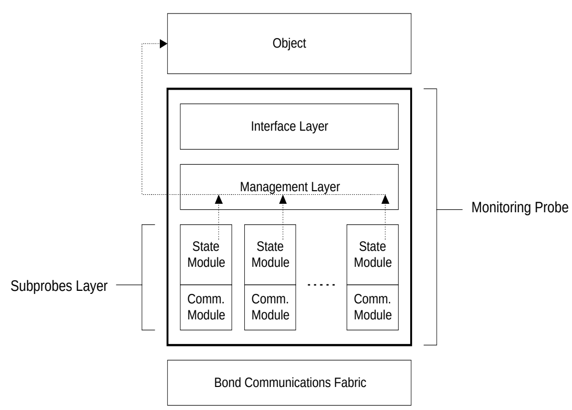

Figure 5: The architecture of monitoring probe. It consists of interface, management, and subprobes layers. Each subprobe corresponds one monitoring relationship, either monitoring or being monitored. The dotted line represents the object reference for callback or property access.

The sub-probes layer consists of a set of sub-probes. Each sub-probe consists of a *communication module* and a *state module*. The communication module implements the monitoring sub-protocol. It not only handles incoming monitoring messages but also composes outgoing monitoring messages. The configuration of state module depends upon the role of the sub-probe. On the monitor side the module is configured to build the local replica of the subject's properties and update them upon receipt of notification messages. On the subject side it collects the subscribed properties and sends them according to the notification pattern.

The management layer has a *reference* to the object that embeds the monitoring probe. The reference is shared by the state modules in the monitoring sub-probes, represented as dotted line in Figure 5. Using this reference, the subject sub-probe can access the static and dynamic properties of the object and the monitor sub-probe can invoke functions of the object, like *callback* mechanism.

The subject sub-probe replies to incoming subscription messages without any interaction with the object. It collects the subscribed properties using the object reference and sends them back to the monitor. An *access control* mechanism enforces a monitoring discipline. This mechanism is discussed elsewhere.

5. Applications

The monitoring framework presented above has been used to monitor the states of Bond servers providing directory service, authentication, dispatching, and software distribution [6], and to distribute real-time stock quotes provided by a *stock watcher* server.

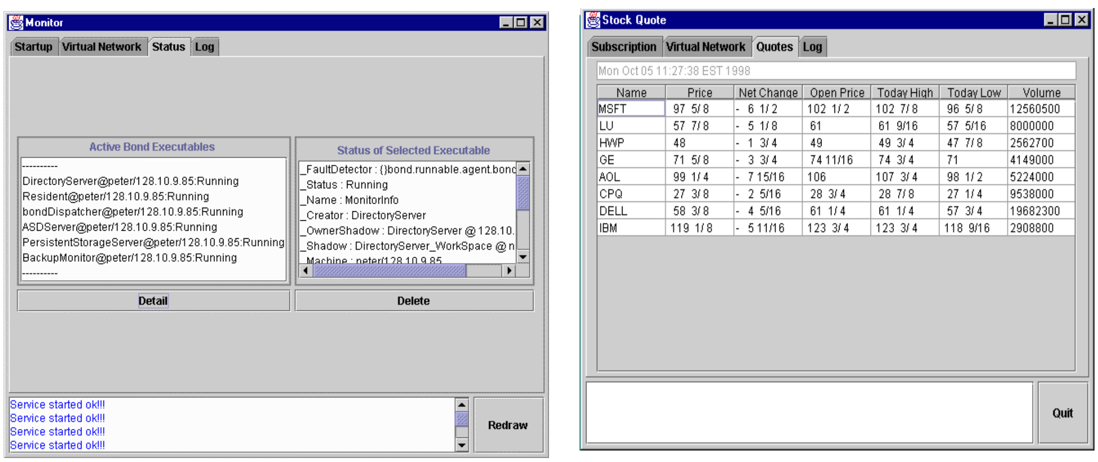

Figure 6: On the left, an interactive display for Bond System Monitor. The right panel shows the detail information of selected server. On the right, an interactive display for the Stock Watcher Client.

The Bond System Monitor is a server responsible for server start-up and shutdown, failure detection, and load balancing among core servers. The servers are subscribed by the system monitor and deliver time-triggered notification during their lifetime. The system monitor identifies the server failures from the missing notification. Each server is required to contain *MonitorInfo*, a mandatory property containing general information like name, address, started time and server-specific information like the number of registered objects of the directory server, the list of logged users of the authentication server, etc. By subscribing this property, the system monitor can obtain detail state of the servers and take actions such as load balancing, as shown in Figure 6. To tolerate the monitor failure the system monitor is replicated so that *backup monitor* subscribes to *primary monitor*.

The Stock-Watcher, SW, is a server maintaining a list of real-time stock quotes and their properties. The SW server sends stock query to a commercial server using an HTTP connection, parses the stock information from HTML-format results, and stores it as dynamic properties. Each property consists of stock name and a collection of related properties such as price, trade volume, etc. as discussed in Section 3. Stock client, an application providing GUI, allows users to specify interesting quotes, desired notification pattern, and manner as well as watch the real-time quotes as shown in Figure 6. Note that the monitoring probe makes it possible to implement the SW server and client without any effort for monitoring implementation, which was the core part of this application.

6. Conclusions

We describe the design and implementation of a subscription-based monitoring model for distributed object systems. We present two applications of the model, the Bond System Monitor and the Stock Watcher. We also have demonstrated that this monitoring model can be successfully applied to fault detection and data source monitoring. By encapsulating the monitoring implementation into monitoring probe, we can dynamically attach it to application objects reducing development overhead.

7. Acknowledgments

The work reported in this paper is partially supported by the National Science Foundation grants BIR-9301210 and MCB-9527131, by the California Institute of Technology, under the Scalable I/O Initiative, by the Intel Corporation, and by the Computational Science Alliance and the NCSA at the University of Illinois.

8. References

[1] I. Foster and C. Kesselman. The Globus Project: A Status Report. In *Proc. IPPS/SPDP '98 Heterogeneous Computing Workshop*, pg. 4-18, 1998.

[2] T. Finin, et al. Specification of the KQML Agent-Communication Language. DARPA Knowledge Sharing Initiative draft, 1993.

[3] K. Chandy, A. Rifkin, and E. Schooler. Using Announce-Listen with Global Events to Develop Distributed Control Systems. In *Proc. ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998.

[4] P. Stelling, I. Foster, C. Kesselman, C. Lee, and G. Laszewski. A Fault Detection Service for Wide Area Distributed Computations. In *Proc. 7th IEEE Symp. on High Performance Distributed Computing*, 1998.

[5] L. Boloni, R. Hao, K. Jun, and D. C. Marinescu. Subprotocols: an object oriented solution to semantic understanding of messages in a distributed object system (submitted).

[6] K. Jun, L. Boloni, R. Hao, and D. C. Marinescu. Bond system monitor. Technical Report CSD-TR #98-026, Purdue University, 1998.

[7] K. Chandy. Caltech Infospheres Project Overview: Information Infrastructures for Task Forces, 1996. http://www.infospheres.caltech.edu/papers/infosummary/infosummary.html.

[8] L. Boloni, K. Jun, M. Sirbu, and D. C. Marinescu. Seamless Metacomputing with Bond. Technical Report CSD-TR #98-010, Purdue University, 1998.
[9] R. Schlichting and F. Schneider. Fail-Stop Processors: an Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computer Systems* 1(3): 222-238, 1983.
[10] R. Englander. *Developing JAVA Beans*. O'Reilly, 1997.
[11] A. Grimshaw, A. Nguyen-Tuong, and W. Wulf. Campus-wide Computing: Early Results Using Legion at the University of Virginia. Technical Report CS-95-19, University of Virginia, 1995.