Reflective Middleware for Integrating Network Monitoring with Adaptive Object Messaging

Qi Han, Sebastian Gutierrez-Nolasco, and Nalini Venkatasubramanian
University of California-Irvine

Abstract

In the future, applications will need to execute in a ubiquitous environment with varying network conditions (connectivity, bandwidth, etc.) and system constraints (e.g., power and storage). The distributed object paradigm is often used to facilitate the development of large-scale distributed applications. However, the traditional object messaging layer operates with limited awareness of underlying system and network conditions, whereas current system and network monitoring tools operate at the network layer with little awareness of application-level object communication requirements. This article explores the possibility, mechanisms, and benefits of filling the gap between object messaging and system monitoring. We introduce connection abstraction as the mechanism for these two layers to communicate and exchange information. Through this integration, object messaging can proactively adapt to changing system conditions; system monitoring policies and parameters can be optimized based on interobject communication properties.

Recent advances in networking and device technologies such as wireless communication, mobile computing, and real-time system support have enabled a new class of applications that require ubiquitous access to information anywhere and anytime. Such distributed applications demand a high degree of flexibility and adaptability in order to deal with:

- Changes in application requirements
- Dynamic changes in the computational and communication environment

Over the years, the distributed object paradigm has gained significant popularity in facilitating the rapid development of large-scale distributed applications. In the distributed object paradigm, application components are modeled as objects that are distributed over a network, and application-level communication is implemented as messages to these distributed objects.

Distributed applications (e.g., streaming high-quality multimedia, real-time target tracking with sensors) are associated with end-to-end quality of service (QoS) requirements: timeliness, reliability, and security. These QoS requirements are implemented as services operating on distributed objects and as layered protocols for object communication (e.g., encrypted messages to objects for security, prioritized messaging for timeliness, and reliable messaging for fault tolerance). However, traditionally the object messaging layer operates with limited awareness of underlying system and network conditions. Similarly, current system and network monitoring tools operate at the network layer with little awareness of application-level object communication requirements.

This article attempts to bridge the gap between the networking and messaging layers by bringing system and network monitoring support to the level of distributed object communication using reflective middleware techniques. Specifically, we introduce the notions of:

- *System-aware messaging*, where changes in system status can proactively trigger object-level communication adaptation
- *Application-aware network monitoring*, where interobject communication properties trigger optimization of monitoring policies and parameters

Such integration brings with it several benefits. First, exposing distributed/global knowledge of system state to the object messaging layer can allow the application level to transparently adapt to changes in system state proactively. Typically, conditions such as network congestion are detected at the application layer only after the application has experienced QoS degradation. With a priori information on the trend of system state, applications can adapt prior to experiencing QoS failures. Second, object-level communication services may be bypassed if network state is available to the object messaging layer. For instance, if one anticipates low network load and low packet drop rates, the object messaging layer can employ low overhead reliability and timeliness protocols. Similarly, encryption for secure messaging in mobile environments may be eliminated if the host and target objects reside on nodes that are collocated in a secure domain; such node location information is usually maintained at the network management layer. Finally, system and network monitoring services can be improved with knowledge of application requirements. For example, differential monitoring of network domains based on application needs can be enforced if the dominant communication requirement (e.g., timeliness, reliability) varies; such communication requirements are usually known by the object messaging layer. Optimizations such as those described above

0890-8044/04/$20.00 © 2004 IEEE

can help significantly to conserve resources and power at the endpoints while reducing message processing overhead and delays in the network.

Bridging Network Monitoring and Object Messaging

Distributed object applications are designed as a set of collaborating objects distributed over a network of processing nodes, interacting only via message passing. The object messaging layer is responsible for setting up logical connections between hosts and objects, and communicating directly to low-level communication packages, the message passing paradigm thus provides a powerful abstraction and interface that shields programmers from low-level communication details. While ease of use is clearly gained, it is difficult to assess the performance and overhead of object messaging without awareness of the underlying network characteristics. Identifying communication patterns in terms of message size and frequency, and estimating communication performance based on current network characteristics is of utmost importance for the tuning of communication protocols. However, in highly dynamic environments, available network resources, connectivity, and communication characteristics (e.g., reliability or security) change very quickly; these changes must be monitored and conveyed to the messaging layer.

Currently system and network monitoring is done with little knowledge of the application communication requirements. In other words, network status is tracked under the assumption that applications use the network uniformly (e.g., real-time applications have similar timeliness requirements or are best effort; either all applications require timely service or none do). Maintaining accurate system state information is expensive since network and system components must be probed frequently and the changes maintained in a repository. This introduces additional network traffic; large numbers of updates to a repository (database, directory service) can cause significant locking overhead. While differentiated services supply individualized bandwidth guarantees to applications by classifying applications into coarse-grained classes (guaranteed and best effort), system monitoring must still pay the cost of unnecessary monitoring overhead without actually satisfying application requirements. Customizing or finetuning network monitoring based on application characteristics can help ensure that most applications receive information at the desired level of quality while minimizing resource consumption.

Therefore, in order to serve applications better, a mechanism needs to be developed for the network monitoring and object messaging layers to exchange information. We introduce the connection abstraction as an interaction mechanism between these two layers. Information exchange between the two levels occurs via this abstraction without loss of separation of concerns. The messaging layer interprets the connection as an object-to-object mapping, while the networking layer views the connection as a node-to-node mapping. System monitoring middleware components are used to maintain and deliver network status information and connection data (latency, loss rate, and load) at a desirable level of accuracy; the object messaging layer implements messaging adaptations based on current load and connectivity properties (Fig. 1). Similarly, the messaging middleware provides application meta-data (e.g., application requirements) the system monitoring middleware may use to adapt its monitoring policies and reduce its management overhead.

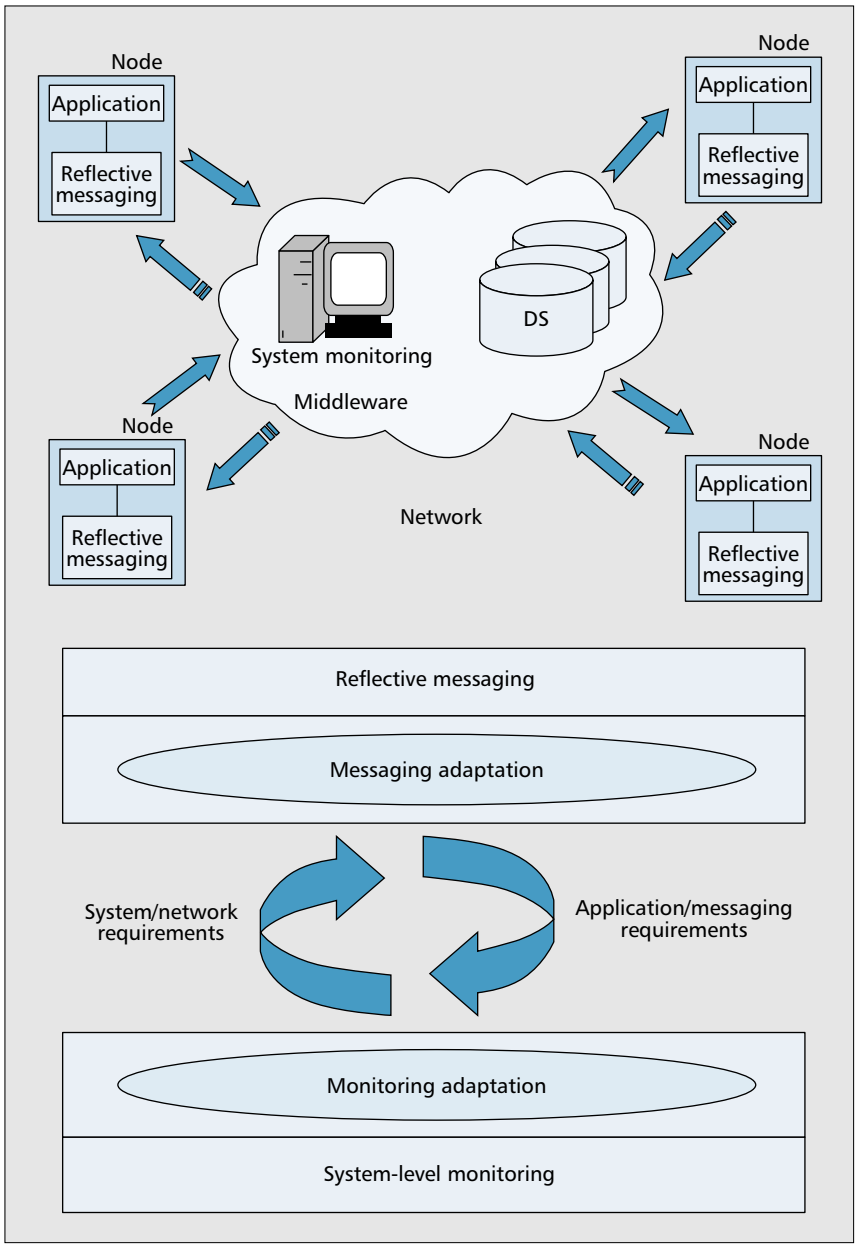

Figure 1. *A reflective architecture for bridging system monitoring middleware and reflective object messaging.*

The Connection Abstraction

In order to provide customization of messaging services and network monitoring, we develop the connection abstraction, which is capable of:

- Mapping data collected at the network level and translating it into useful information for object messaging

C_{id}: The connection identifier.

C_{eplst}: The endpoint list specifies the pair of nodes/objects (source,target) that share the logical connection.

C_{kind}: Connections are classified based on expected duration. Long-lived connections are tagged *persistent*, while short-lived connections are tagged *transient*. Connections may also be *mediated* if requiring QoS enforcement.

C_{req}: The set of communication requirements the application is interested in maintaining, such as security and timeliness.

C_{domain}: The network domain characteristics (if they exist), such as secure.

$C_{latency}$: The latency of the connection.

$C_{loss\text{-}rate}$: The average transmission loss rate.

C_{load}: The current connection workload in terms of average transmission frequency and size.

$Inode_{density}$: The aggregated load from all the connections between two nodes.

Figure 2. *Connection annotations.*

- Mapping application communication requirements to network parameters that allow finetuning of the network monitoring process
- Capturing relevant information about the behavior of the connection

The connection is a virtual link between two communicating objects distributed in the system, and works as an interaction mechanism between the system monitoring and object messaging layers. At the system monitoring level, connection endpoints refer to nodes on which objects reside. The connection provides application-specific information that helps specialize and further refine the system monitoring process. At the messaging layer level, the connection provides feedback regarding current system and network status, which is used to customize and finetune messaging protocols to achieve better overall application communication performance.

Since a connection is between distributed objects, there might be several connections between the same pair of nodes, each with possibly different communication requirements. Note that it may be useful to pre-establish predefined connections between nodes and try to exploit similar communication requirements between objects or implement independent performance optimizations of the communication protocols used at the messaging layer. A connection holds a variety of system- and application-oriented information; a set of connection annotation is defined as in Fig. 2.

Formally, we define the connection as a triplet v_1, v_2, $aparam(u,l)$ defining the source and target node as well as the adaptation parameters to be monitored with their corresponding (upper and lower) threshold values. We divide the connection management mechanism into three phases, according to connection functionality, as follows:

Initialization: A connection may be statically created at startup or dynamically created when a message is sent between objects for the first time. In either case, the initialization specifies the relevant parameters to be monitored and threshold values for these parameters that determine when a notification event should be triggered.

Monitoring and Adaptation: Once the connection has been initialized, system monitoring will track the connection parameters. When a parameter threshold is violated, an event notification is sent to the endpoint nodes, which may respond with a suitable messaging adaptation. This adaptation may also trigger new parameters to monitor or modify the current threshold values. This process of connection monitoring and adaptation is repeated until the messaging middleware tears down the connection.

Finalization: When the connection is no longer desired or needed it is removed to free the monitoring measures at the system and network levels.

In the next two sections we describe in detail how the connection abstraction links the system monitoring and reflective messaging middleware.

The System Monitoring Middleware

In this section we describe the system monitoring middleware architecture (Fig. 3) that:

- Facilitates the integration of system monitoring and object messaging
- Provides seamless access of underlying system conditions to both system components and end users

To achieve this, the system monitoring middleware architecture must handle requests from the object messaging layer without ignoring or delaying important status updates from nodes or links.

In the future, large-scale ubiquitous computing environments will consist of diverse applications executing on heterogeneous devices, systems, and networks. Managing the evolution of such large-scale systems requires efficient repositories of system- and application-level information that can be used to allocate resources effectively, and provide application requirements such as reliability, security, and QoS. Such information repositories, also termed *directory services* (DS), will form the crux of effective middleware for dynamic distributed events. Directory services hold system, user, and object-level information that can be used by various middleware components for object messaging, resource provisioning, security, and location management. DS provide seamless access of this information to both system components and end users.

Three basic components (data collection, request management, and event notification) are used for communicating with messaging middleware.

The data collection module: This module gathers status information from sources including nodes (mobile/fixed hosts) and network links. These sources can be programmed to send out information updates periodically, respond to value requests, or send out notifications when their values are

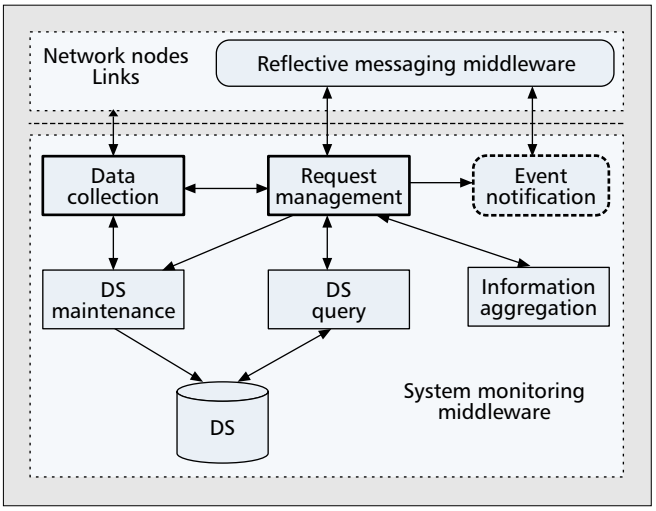

Figure 3. *The system monitoring middleware architecture.*

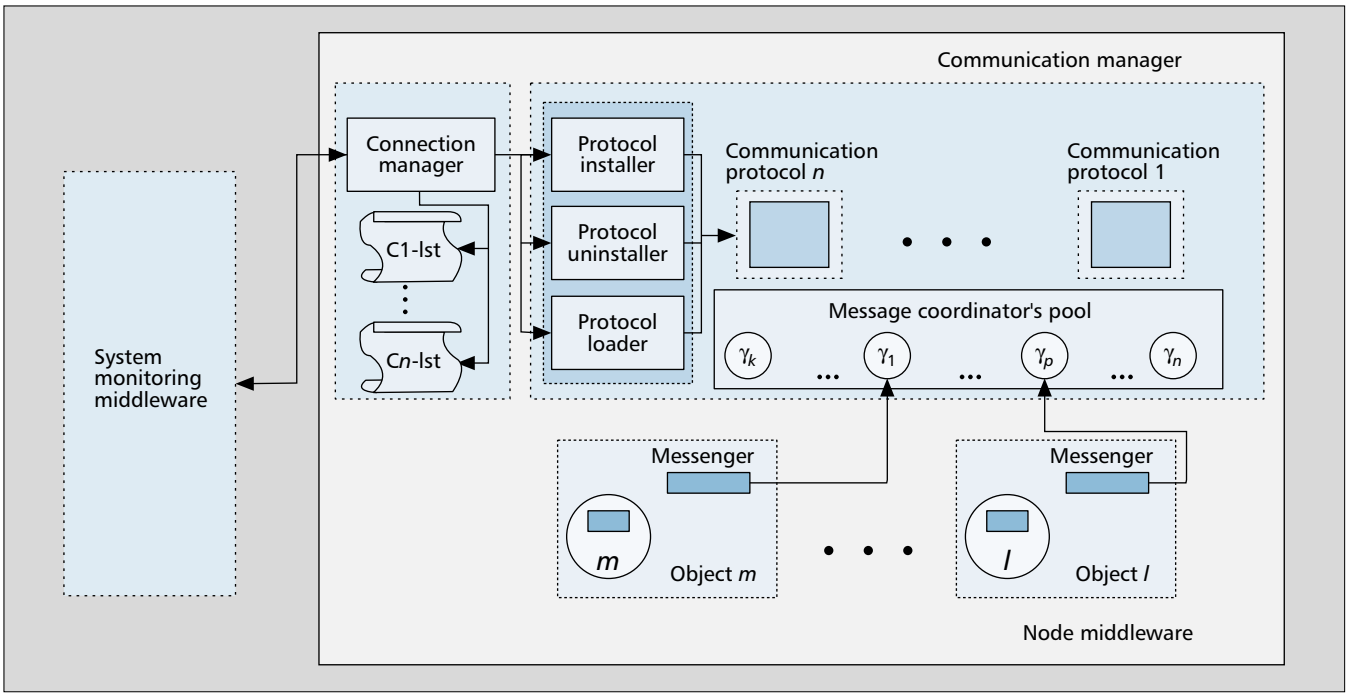

Figure 4. *The reflective communication framework model.*

beyond a certain prespecified threshold. The obtained raw data is sent to the *DS maintenance module*, which decides whether or not to modify the current stored values in the DS.

The request management module: This module accepts requests from messaging middleware. These requests are classified into information retrieval and information registration. Information retrieval requests are read-only requests that obtain the current status of the system. The request management module may retrieve the raw data (through *the DS query module*) or derived/aggregated connection data (via *the information aggregation module*). Information registration requests, on the other hand, are write requests that register connection attributes and adaptation parameters (with specified triggering conditions) to the monitoring layer. In addition, the request management module influences the DS maintenance process based on connection-specific requirements.

The event notification module: This module is specifically used for processing connection-oriented information registration requests from messaging middleware. When the threshold of the registered adaptation parameter for a connection is reached, a notification is sent to messaging middleware. This is used to trigger adaptations of messaging protocols according to the changes in underlying system and network conditions.

We discuss specific suitable system monitoring policies to facilitate timely response to changes in system conditions.

Reflective Messaging Middleware

Distributed applications have communication requirements that need to be provided independent of the underlying network characteristics. Usually, this is achieved by messaging protocols implemented by the object messaging layer. In a reflective messaging layer, we model a system as a composition of two kinds of objects, base objects and meta objects, distributed over a network of processing nodes. Base objects carry out application-level computation, while meta objects customize the communication of base-level objects. Meta objects communicate with each other via message passing as do base objects.

In order to provide customization of messaging protocols in a transparent and scalable fashion, we developed a reflective communication framework (Fig. 4), where each base level object has a meta-level object, called a *messenger*, that serves as the customized and transparent message queue for that base-level object. There is one *connection manager* and one *communication manager* in every node of the distributed system. The connection manager collects relevant information regarding logical connections between its node and (potentially) any other node of the system. Thus, every connection manager holds a set of lists (C_1-*lst* – C_n-*lst*) representing all the current open logical connections of the node. The communication manager implements and controls the correct composition of communication protocols specified by messengers on that node.

The communication manager has a set of communication protocol objects, each implementing a particular messaging protocol provided by the framework (e.g., reliable, timely, secure). This scheme allows us to abstract a core set of messaging protocols and share it between the different messengers on a node, simplifying the synchronization and composition process. Furthermore, it encourages separation of concerns in the process of message transmission and reception.

In purely reflective architectures, reasoning about the semantics of communication composition may be complicated; moreover, its implementation may be inefficient. In order to maintain accurate semantics and provide an efficient implementation of the reflective architecture, the communication manager implements a set of meta-level entities, called *communication message coordinators* (or simply *message coordinators*).

Every message sent is intercepted by the messenger, which consults the connection manager to extract the protocols supported by the connection and determine if the message needs to be processed with additional messaging protocols. If so, the communication manager assigns a message coordinator to the message, who handles the composition of the messaging protocols requested. The message coordinator ensures the correct order of composition of required protocols and provides a coordination mechanism between the messenger and the protocols that provide it. This concept of reusable message coordinators is an efficient way to handle the service request of

each messenger without having to pay the bottleneck associated with the centralization of the services in the node communication manager and allows us to process multiple messages concurrently. At the end the processed message is returned to the messenger, who sends it through the corresponding connection. Handling messages delivered at the receiver end is simple since the specific messaging protocols applied to the message payload must only be unwrapped.

Application-Driven Monitoring and System-Aware Messaging

This section describes in detail how the system monitoring and reflective messaging middleware cooperate to optimize system performance by:

- Boosting application satisfaction
- Controlling monitoring and messaging overhead

The system monitoring middleware implements proactive network monitoring strategies to improve DS quality, while the reflective messaging middleware takes advantage of system meta-data provided by the system monitoring middleware to reduce communication overhead.

Adaptive System Monitoring

The system monitoring layer collects system and network status parameters from network devices, and processes and maintains the following information in the DS:

- Link parameters: This includes residual link bandwidth, end-to-end delay on links, link load, and link packet drop rate.
- Node parameters: This includes CPU utilization, buffer capacity, disk bandwidth, location (in terms of domain or cell), connectivity, power constraints, and throughput.

Keeping track of system conditions in dynamic environments is challenging due to the following reasons.

Sheer volume of streaming data: Due to the dynamicity of the system and network, the amount of related information that needs to be captured and stored rapidly and accurately is huge and often this information needs to be processed in real time.

Tedious information aggregation: Given the enormous amount of links, aggregating connection information from link-level data is difficult, and customizing the aggregation and mapping process for specific applications can be very tedious. End users requiring access to this dynamically changing information present variable requirements in terms of timeliness, security, or reliability of the service and expectations of data precision and freshness.

Need for monitoring to be unobtrusive: Network and service providers would like to ensure effective utilization of underlying computation, communication, and storage resources. Ideally, we ensure that applications receive information at their desired level of accuracy while minimizing resource consumption introduced by monitoring tasks.

System Monitoring for Dynamic Environments — System monitoring involves two steps:

- Data sampling where system and network level information is obtained from nodes, routers, switches, and so on. The frequency of sampling may be fixed or varied over time
- Data updating where the DS is updated based on current sampled values

The effectiveness of system monitoring and the overhead incurred is dependent on two factors: data sampling frequency and data representation. By observing the trend of value changes, sampling frequency can be adjusted. For example, if the value is relatively stable, there is no need to sample very frequently.

Different approaches have been presented and evaluated to explore the pros and cons of various data representations [1]. The simplistic *instantaneous value-based approach* samples sources regularly; the DS is updated with the collected exact values. The sampling period solely determines the accuracy of the information stored in the DS. In highly dynamic traffic, the sampling has to be done at a very high frequency to prevent information from being outdated.

A *static interval-based approac*h can be used to reduce sampling and DS update overhead. It defines a fixed interval that is used to partition the capacity of the collected information into a fixed number (say n) of equal sized classes. The classes are represented by corresponding indices 0, 1, 2, …, $(n - 1)$. A probe is initiated at each sampling interval to obtain current information from the managed entities. If the obtained value is out of the interval indicated by the current index, the DS is updated with another index; otherwise, no update is needed.

The static interval-based approach incurs less overhead than the instantaneous value-based approach as frequent DS updates are not necessary, but it is not flexible since the interval size is fixed. To address this issue, a family of dynamic-range-based approaches are proposed to modify the range dynamically based on sampled information. A throttle-based [1] approach uses the average value of samples in a previous monitoring window to decide whether a range adjustment is needed or not. A more analytical approach using time series based technique [2] has been proposed; the objective here is to predict a range for a future interval such that the deviation between the predicted and observed values remains within a given confidence level. Based on the size of the range and the confidence level, a bound on the sampling rate is determined. The range as well as the sampling rate are then dynamically adjusted based on the burstiness of the incoming traffic. Another approach analyzes the cost involved in the whole process of monitoring and derives an optimal condition to ensure minimized cost [3].

System Monitoring Scenario 1: Selective Maintenance of Host Mobility — In mobile environments, constant mobility of end users could cause significant variation in network resource availability; this complicates network monitoring. While keeping track of individual user mobility patterns can help network monitoring, maintaining accurate location information for each user in the DS can entail very high overhead. Recent work [4] has shown the effectiveness of maintaining aggregate user mobility information (the mobile host population in a certain region at a certain time) in the DS to finetune network monitoring parameters. This information can be obtained from wireless access points to the fixed network, thereby eliminating the need for constant monitoring of individual mobile host locations. In future networks, fixed host and mobile hosts will coexist. Monitoring policies must be selected based on the degree of host mobility as indicated by respective node parameters. For example, it is unnecessary to apply aggregate-mobility-driven system monitoring if many of the hosts in the monitoring region are static. Furthermore, if the possibility of certain hosts (with security and QoS requirements) moving out of a current network domain is very high, keeping track of individual host mobility is necessary since aggregate host mobility only provides coarse-level information.

Finetuning System Monitoring for Object Messaging — Application-aware system monitoring:

- Provides system meta-data (especially connection-oriented information) to reflective messaging middleware

- Adapts monitoring policies and parameters based on application meta-data and interobject communication properties

In the following paragraphs we describe system meta-data supplied by system monitoring middleware and the mechanisms to derive this information. Network domain characteristics (C_{domain}) can be retrieved directly from the DS, since the DS maintains a mapping between a node and its current domain. C_{domain} is often queried during the connection initialization phase and helps reflective messaging middleware to decide if a new set of communication protocols needs to be installed or not. If the node moves out of the current domain, a notification may be sent to the messaging layer if necessary for possible messaging adaptation.

Calculating parameters such as connection latency $C_{latency}$, packet drop rate $C_{loss\text{-}rate}$, load C_{load}, and internode density $Inode_{density}$ is more involved since they are assimilated from the raw data in the DS, where this information is maintained for individual links and nodes. Let us assume that between two endpoints of a connection C, there are n feasible paths (e.g., shortest widest path); path p^i consists of m_i links (i.e., $C = \{p_1, p_2, \ldots, p_i, \ldots p_n\}$, $p_i = l^1 \rightarrow l^2 \rightarrow \ldots \rightarrow l^j \ldots \rightarrow l^{m_i}$). A connection could use any one of the paths. We derive path parameters from the link parameters and then infer connection parameters from the path parameters based on communication attributes (C_{req}) of applications as follows:

- $C_{latency}$: The path latency is the total latency of the links along that path,

$$p^i_{latency} = \sum_{j=1}^{m_i} l^j_{latency}.$$

  Connection latency is the lowest path latency if C_{req} = *timeliness*; otherwise, it is the average or random path latency.
- $C_{loss\text{-}rate}$: The packet drop rate of a path is the maximum link packet drop rate along the path, $p_{loss\text{–}rate} = max(l^j_{loss\text{-}rate})$, $j = 1, \ldots, m_i$. The connection packet drop rate is the highest path packet drop rate if C_{req} = *reliability*; otherwise, it is the average or random path packet drop rate.
- C_{load}: A path load is the maximum link load of all the links along the path: $p^i_{load} = max(l^j_{load})$, j = 1, …, m_i. Connection load is the average path load, $C_{load} = avg(p^i_{load})$, $i = 1, \ldots, n$.
- $Inode_{density}$: It is the sum of the load of all connections between the same endpoints.

At the connection initialization phase, these parameters are requested by reflective messaging middleware as preliminary guidelines for establishing the connection. Triggering conditions for these parameters are then passed to system monitoring middleware, thus initiating background monitoring. When a significant change is observed or a threshold is reached for either of these adaptation parameters, an event notification is sent to reflective messaging middleware. Further adaptation of the connection is started and new triggering conditions are passed to system monitoring middleware. The above process allows messaging middleware to adapt object communication more intelligently.

At the same time, system monitoring mechanisms can take advantage of application knowledge and interobject communication properties to finetune the monitoring process. In the following paragraphs, we explain the application meta-data provided by messaging middleware and how this data is used by system monitoring. Three connection attributes (C_{eplst}, C_{kind}, C_{req}) are used for system monitoring middleware to infer interobject communication properties and application workload, which will then be used for customization of monitoring policies and parameters as follows:

***C_{eplst}*:** With knowledge of connection endpoints (nodes), we can derive the notion of connection density, which is the number of connections between two specific nodes. Connection density information can help determine if monitoring should be tightened or relaxed (through adjustment of sampling frequency and range size) for a link or node. Maintaining C_{eplst} can also help to identify whether or not two connections share the same two endpoints. If two connections are established from the same source node to the same target node, connection properties such as $C_{latency}$, $C_{loss\text{–}rate}$, and C_{load} for one connection can be reused for another, thereby eliminating additional information request messages and network probes.

***C_{kind}*:** This parameter is used by system monitoring middleware to infer stability of allocated system resources. For example, C_{kind} = *mediated & persistent* indicates that a specific amount of data transmission is planned and resources have been reserved. One may also infer that the allocated network and resources will be consumed for a certain period of time; hence, resource availability status will be stable for that period. The monitoring can therefore be relaxed for that part of the network.

***C_{req}*:** Together with C_{eplst}, this helps system monitoring middleware to detect the dominant communication requirements (timeliness, reliability, or security) between two nodes, based on which monitoring policies can be tailored or switched (e.g., to ensure timely collection of information).

These three connection attributes are passed from the messaging layer to the system monitoring middleware after the connection is established.

System Monitoring Scenario 2: Proactive Maintenance of Time-Sensitive Information — With the emergence of large-scale ubiquitous environments, providing seamless timely access to dynamically changing data (e.g., sensor data) is gaining importance. The DS must reflect changes in the environments as closely as possible; this process can consume significant computation and communication resources, thereby causing violations of real-time requirements. While approaches such as those described above have been developed to address the accuracy-cost trade-off, further research is required to incorporate timeliness into the monitoring process. Real-time database researchers have investigated issues in maintaining data accuracy while ensuring transaction timeliness. Recent work [5] has addressed the timeliness/accuracy/cost trade-off in maintaining directory services for real-time applications; algorithms are developed that exploit the accuracy and latency margins to ensure that most applications receive information at the desired levels of quality and timeliness while minimizing resource utilization.

We enhance the monitoring process given knowledge of application timeliness requirements. If the dominant communication pattern indicates time-sensitive service is desirable for a certain monitoring region, real-time system monitoring strategies should be applied for that region without affecting the monitoring techniques for other regions. Note that in this case latency information should be monitored more closely than other information.

Adaptive Messaging

The reflective messaging layer customizes messaging protocols to adapt to changing network and system conditions. This is achieved by finetuning messaging protocols using the information provided by the system monitoring layer and the application communication requirements. For instance, many applications use time-sensitive information to ensure application accuracy (e.g., real time target tracking) or provide end-to-end QoS (e.g., streaming high-quality multimedia). However, it is difficult to ensure continuous flow of time-sensitive information due to the heterogeneity and dynamicity of the underlying network. Even for high performance networks,

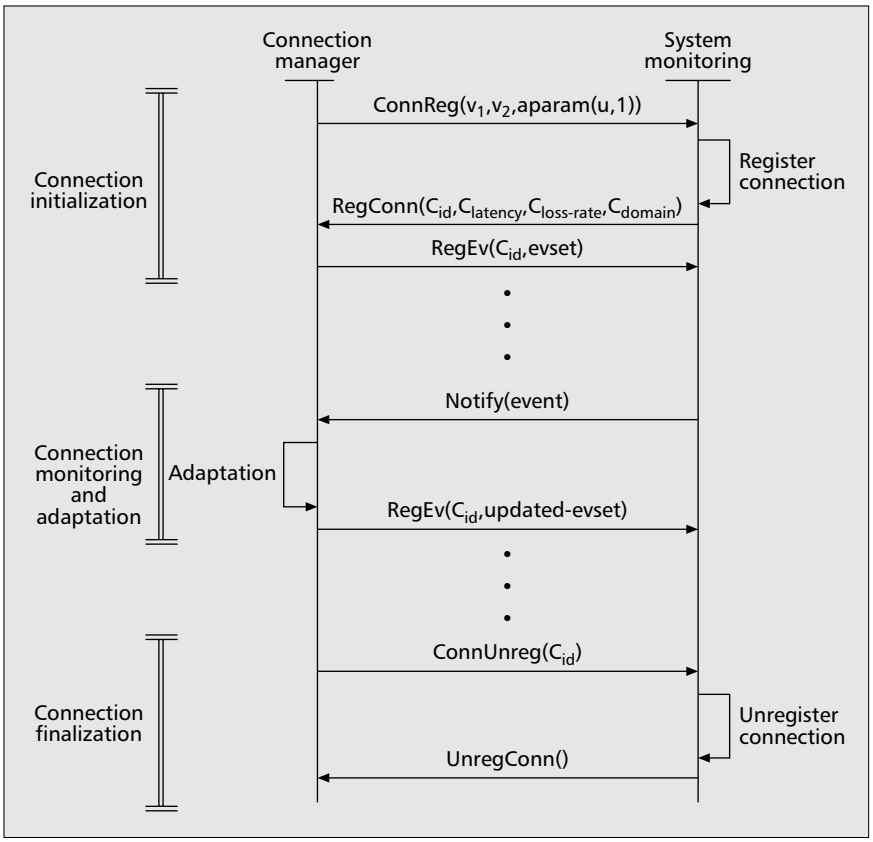

Figure 5. *A event diagram depicting exchange of information using the connection.*

latencies vary from 20–200 ms and bandwidth between 0.5–5 Mb/s. Since object messaging frameworks are not aware of the current network status, extended transient network loads are indistinguishable from a network partition or failure. As a result, the application will experience sudden degradation of communication quality.

Let us assume that the application is capable of specifying a set of value ranges, called *tolerance parameters*, that define the degradation or adaptation range allowed by the application. For example, key length is a tolerance parameter for a secure protocol; the acknowledgment type (positive or negative) as well as sender/receiver window size and maximum number of retransmissions are tolerance parameters for a reliable protocol at the message layer. If the tolerance parameter is not specified, we assume that best effort behavior is implied. Object messaging uses this information to create and customize connections to provide the best possible application satisfaction attainable given the current network and system status.

Figure 5 depicts the exchange of messages between the connection manager at the node and system monitoring during the three phases of connection management. An initial connection registration message *ConnReg*(v_1, v_2, *aparam*(u,l)) is sent to the monitoring layer, where v_1,v_2 are the endpoint of the connection, and *aparam*(u,l) is the set of adaptation parameters with specified upper and lower values that define the threshold values. Once the connection has been registered, the connection manager receives the connection initial status in a message of the form *RegConn*(C_{id},$C_{latency}$,$C_{lossrate}$, C_{domain}) specifying the connection identifier, initial connection latency, drop rate, and domain characteristics, respectively.

Using this information, application protocols are selected to match the tolerance requirements, and monitoring parameters thresholds are determined. The connection manager then registers the set of events of interest (*evset*) by sending a message of the form *Regev*(C_{id},*evset*) to system monitoring. System monitoring notifies the connection manager when a range violation occurs (*notify(event)*) , which may trigger an adaptation procedure. A message *Regev*(C_{id},*updated–evset*) is sent to system monitoring to update the monitoring parameters (if needed). This process of connection monitoring and adaptation is repeated until teardown, when a message of the form *ConnUnreg*(C_{id}) is issued to the system monitoring. A termination confirmation message of the form *UnregConn()* to the connection manager and resources are deallocated.

We now illustrate the monitoring and adaptation phase using a series of examples to show how communication finetuning is achieved.

Object Messaging Scenario 1: Reliable Timely Delivery — Let us assume that the application defines a message delivery deadline in order to facilitate the flow of time-sensitive information. This implies that the object messaging mechanism must deliver a message before reaching the end of the time span; otherwise, the message is discarded. Given these requirements, object messaging selects a reliable timely delivery protocol with timeouts, positive acknowledgment, and selected retransmissions. After receiving the connection status message *RegConn*(C_{id},$C_{latency}$,$C_{loss\text{-}rate}$,C_{domain}), the connection manager at the node specifies connection latency and loss rate as the parameters to be closely monitored. If there is a sudden increase in the connection loss rate or latency, the connection manager will select a negative acknowledgment mechanism and increase the number of retransmissions. If the situation does not improve, the connection manager will expand the time span of the message to the maximum time allowed by the application (i.e., initially the message time span is less than the delivery deadline), thus limiting the number of discarded messages at the target endpoint.

Object Messaging Scenario 2: Secure Messaging in Mobile Environments — In this example we assume that the network is partitioned into geographical domains with varying security levels. Mobile nodes ensure that messages traveling in domains with inadequate security are suitably encrypted. Initially, we assume that both endpoints of a message coreside in a secure domain; hence, no encryption is necessary. When nodes move, messages may cross domains; domain borders must be identified in advance to implement dynamic reconfiguration of communication protocols when a target node moves out from a secure domain. In this particular case, the system monitoring middleware will notify the connection manager (of both endpoints) with a domain change event that triggers the installation of a secure protocol S_1 to guarantee message integrity, secrecy and authenticity at both endpoints. The protocol installer uses the protocol loader to load protocol S_1 and notifies the connection manager that the connection protocol list has been updated. Note that this event will be triggered only if the application specified the security requirement a priori to the monitoring layer.

Object Messaging Scenario 3: Secure Timely Delivery in Mobile Environments — An interesting situation arises if we add the security requirement to our previous time-sensitive example. The connection manager must determine how to layer the security and timeliness protocols. Intuitively, layering the reliable timely delivery protocol on top of the security protocol will not work because the composite protocol may travel through an insecure medium, which may corrupt or fake messages. For instance, the header of the protocol will remain unprotected and may be modified. However, layering the security protocol on top of the reliable timely delivery protocol may not work either since encryption is an intensive computational task that may significantly alter the message timing parameter. Furthermore, most security protocols work under the assumption of session-based communication where frequent rekeying operations must be executed. Thus, messages may face temporary delays due to rekeying, contributing to further communication overhead.

A solution to this problem will involve careful selection of protocol implementation tuning parameters to ensure correct protocol composition. Such adaptation can be greatly optimized with knowledge of network latencies and security parameters.

Implementation and Performance Issues

The adaptive communication framework presented in this article is integrated into a runtime layer under the CompOSE|Q middleware architecture [6]. This framework consists of a set of runtime kernels that reside on individual nodes of the distributed system and a set of components that provide four basic core services — remote creation, dissemination services, distributed snapshot, and directory services — to the application layer. Applications executed in this environment are modeled as distributed objects that communicate (using the adaptive communication framework) with each other via asynchronous message passing; the adaptive communication framework collects information regarding logical connections between objects, such as type and frequency of communication and protocols required, and customizes messages accordingly.

Application-driven monitoring is done under the AutoSeC framework [1], which aims to support dynamic selection of system monitoring policies for network management applications based on current system conditions and user requirements. AutoSeC is built on top of the CompOSE|Q directory service and uses the directory service to store collected data and supply the information for adaptive communication. Server- and network-related information is collected using Simple Network Management Protocol (SNMP), which is readily available on most servers and routers. In addition, XML is used as a messaging format to communicate between components in a distributed environment.

Our testbed is a network of Sun Ultra5 workstations (333 MHz UltraSPARC IIi with 256 kbytes external cache and 128 Mbytes RAM) running Solaris 2.7 connected via a 10/100 Mb/s Ethernet link and executing the CompOSE|Q runtime. The runtime is implemented in Java and uses OpenLDAP's slapd Lightweight Directory Access Protocol (LDAP) server together with the Berkeley Sleepy Cat Database as the backend of the directory service. slapd is a standalone LDAP daemon that listens for LDAP connections on a port and responds to the LDAP operations it receives over these connections.

In order to evaluate the benefits of application-driven monitoring and system-aware messaging, we explore:

- The relationship between application satisfaction and monitoring overhead
- The reflective messaging overhead
- The performance gain due to message adaptation

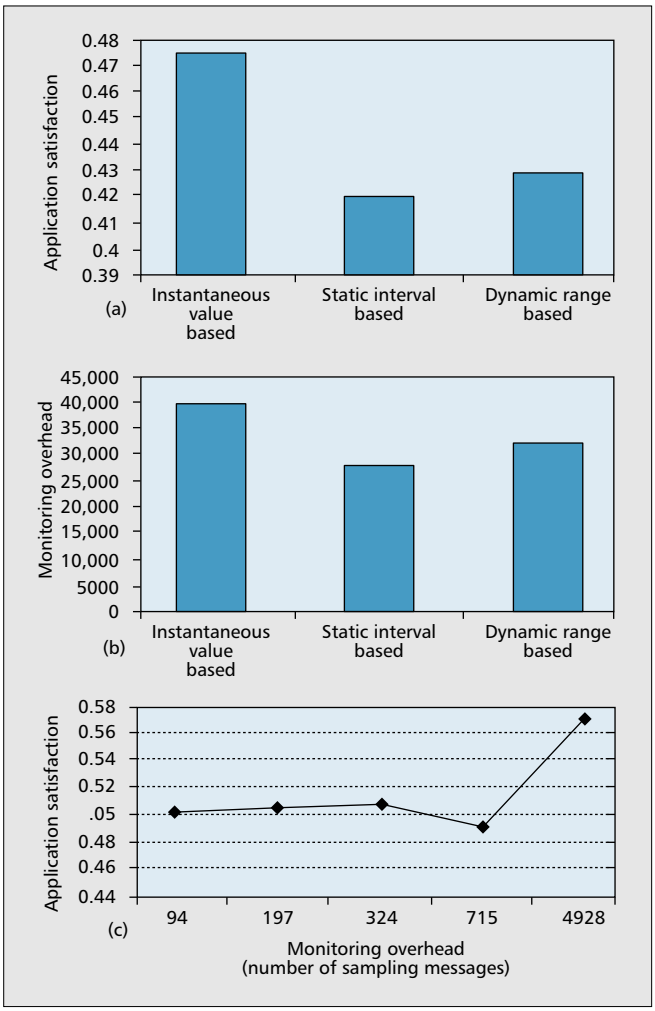

Figure 6. *System monitoring performance issues: a) application satisfaction comparison; b) monitoring overhead comparison; c) application satisfaction vs. overhead.*

The effectiveness of system monitoring is dependent on two factors: application satisfaction and monitoring overhead. Figure 6a shows the application satisfaction obtained under different system monitoring policies (instantaneous value based, static interval based, dynamic range based). Figure 6b demonstrates the monitoring overhead introduced to maintain the directory service in order to achieve the application satisfaction shown in Fig. 6a. Analysis of the performance results indicates that the monitoring policy plays a significant role in application satisfaction and the incurred overhead. Dynamic range-based policies provide a good balance between application satisfaction and incurred overhead. Intuitively we would expect application satisfaction to increase with a rise in sampling frequency (i.e., increasing monitoring overhead); Fig. 6c demonstrates that increasing the monitoring overhead does not always improve application quality, so it is important to adjust monitoring parameters reasonably based on application requirements. One possible reason for this behavior is that increasing sampling frequency blindly without knowledge of connection-specific requirements merely increases the overhead in the network and messaging layers.

We next determine the overhead introduced in the reflective messaging layer by categorizing the different overheads involved in the messaging process (Table 1). First, we measure the message transmission and reception overheads of the

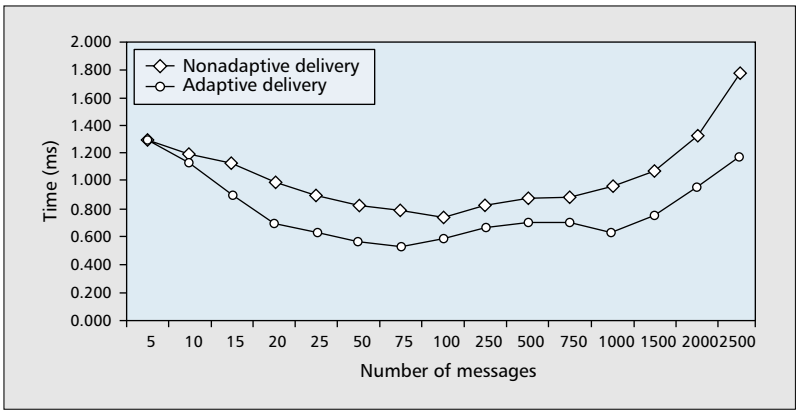

Figure 7. *Object messaging performance issues.*

underlying network. We then measure the time needed to send and receive messages through the reflective messaging layer when no specialized messaging protocols are involved. Finally, we measure the overhead of protocol processing in the reflective messaging layer. Table 1 depicts the send and receive overheads for the single-protocol case (timely delivery) and two-protocol case (secure timely delivery). An obvious performance benefit of system-aware messaging is the information that may lead to the elimination of protocols (e.g., encryption is unnecessary in secure domains).

We can further minimize the messaging overhead by protocol-specific adaptation (Fig. 7). We compare the adaptive and nonadaptive versions of a reliable timely delivery protocol described earlier. The nonadaptive reliable timely delivery protocol does not use any system/network information; its performance is based on the protocol overhead and the parameters specified a priori (e.g.. retransmission scheme, message life span). The adaptive version of the protocol receives feedback from system monitoring and thus is capable of modifying connection parameters in order to adapt to current system and network conditions (e.g., changing from an ack-based retransmission scheme to a nack-based retransmission scheme and increasing the message life span to avoid unnecessary message discards). Note that both curves exhibit similar trends, but the adaptive version responds faster to environmental changes; this helps to improve the overall performance, especially as the number of messages scales up.

Although our performance results are encouraging, we must recognize that the degree of monitoring perturbs the overall performance due to the increasing number of notifications generated when the system scales very rapidly. Ideally, we would like to bound the number of notifications based on available node resources, such as power, processor speed, and memory. Furthermore, protocol installations and uninstallations (triggered by event notification) can consume scarce node resources. Intelligent adaptation techniques that determine when and how to respond to event notifications must be developed. What is required is a set of rules to determine when notification events should be triggered.

Related Work and Concluding Remarks

Existing research on system/network resource monitoring and management has focused on algorithms for cost-effective monitoring to provide reasonably accurate system conditions to applications. ReMoS [7] developed a uniform interface between networks and network-aware applications. It allows applications to discover the properties of their execution environments. The concept of application-aware networking has been developed in the area of active network, where network components are programmed in order to perform customized computation on the user data. Similarly, the Darwin system [8] provides application-specific resource management support by adding resource allocation software into network components without modifying the underlying network architecture. Genesis [9], on the other hand, constructs reflective network architectures to enable reconfigurations of existing network services and modifications of network attributes using dynamic plugins. In comparison, this article attempts to integrate application requirements into system monitoring through the object messaging layer.

Commercially available distributed middleware infrastructures have incorporated the notion of reflection in order to provide the desired level of configurability and openness in a controlled manner. However, they do not deal with interactions of multiple object services executing at the same time or the customization of messaging protocols. In particular, the pluggable protocol framework [10] addresses the lack of support for multiple inter-ORB protocols, and deals with integration and use of multiple ORB messaging and transport protocols. DynamicTAO [11] explores ways to make the various components of an ORB dynamically configurable as well as componentize them to achieve a minimal footprint for small applications [12]. QuO [13, 14] has system condition objects that provide interfaces to resources, mechanisms, ORBs, and so on that need to be observed, measured, or controlled. Delegates reify method requests and evaluate them according to contracts that represent strategies for meeting service level agreements. In the OpenOrb architecture [15–17] every object is represented by multiple models, allowing behavior to be described at different levels of abstraction and from different points of view. Recent research on active objects (called *actors*) has focused on coordination structures and meta-architectures [18]and runtime systems such as Broadway [19] and Actor Foundry [20].These approaches assume point-to-point communication and use the onion skin model as a conceptual foundation for separation of concerns. In this model, each object has a meta level object that defines the semantics of its primitive actions, and multiple protocols may be applied (installed) to a single component object by stacking meta-level objects that implement each protocol.

In recent years, several group and point-to-point communication systems have been developed to provide dynamic composition of messaging protocols [21–26]. Much of this work has focused on mechanisms to provide dynamic composition, installation, and revocation of messaging protocols on the fly. More work is required to develop an adaptation rule base that can be used as an oracle to identify protocols and parameters that match application needs under varying system conditions.

In this article we introduce the connection abstraction as an interaction mechanism between object messaging and system

Messaging overhead	Send (μs)	Receive (μs)
Network	181	141
Unprocessed message	670	607
Time-sensitive messaging	1891	1942
Secure timely messaging	3133	3280

Table 1. *Reflective messaging overheads: message transmission (send) and reception (receive) overheads displayed by category.*

monitoring. We further develop for each layer the supporting techniques to use this connection abstraction. Our proposed integration mechanism provides an appropriate degree of information sharing between the two layers to enable:

- Finetuned object messaging
- Cost-effective system monitoring

Our eventual goal is to provide clean interfaces for integrating adaptation mechanisms at different layers (network, middleware, application, and operating system) while maintaining separation of concerns in layered distributed systems.

References

[1] Q. Han and N. Venkatasubramanian, "AutoSeC: An integrated Middleware Framework for Dynamic Service Brokering," *IEEE Distrib. Sys. Online*, vol. 2, no. 7, 2001.

[2] Z. Fu and N. Venkatasubramanian, "Adaptive Parameter Collection in Dynamic Distributed Environments," *IEEE ICDCS '01*, 2001.

[3] C. Olston, B. T. Loo, and J. Widom, "Adaptive Precision Setting for Cached Approximate Values," *Proc. ACM SIGMOD*, 2001.

[4] Q. Han and N. Venkatasubramanian, "Aggregation Based Information Collection for Mobile Environments," *J. High Speed Networks*, vol. 11, no. 3, 2002.

[5] Q. Han and N. Venkatasubramanian, "Addressing Timeliness/Accuracy/Cost Tradeoffs in Information Collection for Dynamic Environments," *Proc. 24th Int'l. Real Time Sys. Symp.*, 2003.

[6] N. Venkatasubramanian et al., "Design and Implementation of a Composable Reflective Middleware Framework," *Proc. IEEE Int'l. Conf. Distrib. Comp. Sys.*, 2001.

[7] N. Miller and P. Steenkiste, "Collecting Network Status Information for Network-Aware Applications," *Proc. IEEE INFOCOM*, 1999.

[8] P. Chandra *et al.*, "Darwin: Resource Management for Value-Added Customizable Network Service," *Proc. ICNP*, 1998.

[9] M. E. Kounavis *et al.*, "The Genesis Kernel: A Programming System for Spawning Network Architectures," *IEEE JSAC*, vol. 19, no. 3, 2001.

[10] A. Arulanthu *et al.*, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging," *Proc. IFIP/ACM Middleware 2000*, 2000.

[11] F. Kon *et al.*, "Monitoring and Security and Dynamic Configuration with the dynamicTAO Reflective ORB," *Proc. IFIP/ACM Middleware 2000*, 2000.

[12] M. Roman *et al.*, "LegORB and Ubiquitous CORBA," *Proc. IFIP/ACM Wksp. Reflective Middleware 2000*, 2000.

[13] Zinky J, D Bakken, and R Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, 1997, pp. 41–49.

[14] J. Zinky, J. Loyell, and R. Shapiro, "Runtime Performance Modeling and Measurment of Adaptive Distributed Object Applications," *Proc. Int'l. Symp. Distrib. Objects and Apps.*, 2002.

[15] F. Costa, G. Blair, and G. Coulson, "Experiments with Reflective Middleware," Tech. rep. MPG-98-11, Lancaster Univ., 1998.

[16] F. M. Costa and G. Blair, "Integrating Meta-Information Management and Reflection in Middleware," *2nd Int'l. Symp. Distrib. Objects and Apps.*, 2000.

[17] H. A. Duran-Limon and G. Blair, "The Importance of Resource Management in Engineering Distributed Objects," *2nd Int'l. Wksp. Eng. Distrib. Objects*, 2000.

[18] A. Agha *et al.*, "Abstraction and Modularity Mechanisms for Concurrent Computing," *IEEE Parallel and Distrib. Tech.: Sys. and Apps.*, 1993.

[19] D. Sturman, "Modular Specification of Interaction of Interaction Policies in Distributed Computing," Univ. of IL at Urbana-Champaign, 1996.

[20] M. Astley and G. Agha, "Customization and Composition of Distributed Objects: Middleware Abstractions for Policy Management," *6th Int'l. Symp. Foundation of Software Eng.*, 1998.

[21] R. van Renesse, K. Birman and S. Maffeis, "Horus: A Flexible Group Communication System," *Commun. ACM*, vol. 39, no. 4, 1996, pp. 76–83.

[22] M. Garland Hayden, "The Ensemble System," Ph.D. thesis, Cornell Univ., Dept. of Comp. Sci., 1998.

[23] M. Astley, D. Sturman, and G. Agha, "Customizable Middleware for Distributed Software," *Commun. ACM*, 2000.

[24] J. He *et al.*, "Providing QoS Customization in Distributed Object Systems," *IFIP/ACM Int'l. Conf. Distrib. Sys. Platforms (Middleware '01)*, 2001.

[25] S. Gutierrez-Nolasco and N. Venkatasubramanian, "A Reflective Middleware Framework for Communication in Dynamic Environments," *Proc. Int'l. Symp. Distrib. Objects and Apps.*, 2002.

[26] Y. Amir *et al.*, "Scaling Secure Group Communication Systems: Beyond Peer-to-Peer," *3rd DARPA Info. Survivability Conf. and Exposition*, 2003.

Biographies

QI HAN [SM] (qhan@ics.uci.edu) is currently pursuing a Ph.D. from the School of Information and Computer Science at the University of California, Irvine. She received her B.S. in computer science and engineering from Yanshan University, Qinhuangdao, Hebei, China and her M.S. in computer science from Huazhong University of Science and Technology, Wuhan, Hubei, China. Her research interests include distributed systems middleware, mobile computing, and sensor networks.

SEBASTIAN GUTIERREZ-NOLASCO [SM] (seguti@ics.uci.edu) is currently pursuing a Ph.D. from the School of Information and Computer Science at the University of California, Irvine. He received his B.S. in computer engineering from the Instituto Tecnologico Autonomo de Mexico (ITAM). He worked as a research assistant at Cannes Laboratory for Brain Simulation. His main research interests include formal modeling and reasoning, reflective and meta-level architectures, robotics, and neural networks. He is a member of the ACM.

NALINI VENKATASUBRAMANIAN [M] (nalini@ics.uci.edu) is an assistant professor at the School of Information and Computer Science, University of California, Irvine. Her research interests include distributed and parallel systems, middleware, mobile environments, multimedia systems/applications, and formal reasoning of distributed systems. She is specifically interested in developing safe and flexible middleware technology for highly dynamic environments. She was a member of technical staff at Hewlett-Packard Laboratories, Palo Alto, California, for several years where she worked on large-scale distributed systems and interactive multimedia applications. She has also worked on various database management systems and programming languages/compilers for high-performance machines. She has M.S. and Ph.D. degrees in computer science from the University of Illinois, Urbana-Champaign, and is a member of the ACM.