

Reflective Middleware for Integrating Network Monitoring with Adaptive Object Messaging

Qi Han, Sebastian Gutierrez-Nolasco and Nalini Venkatasubramanian
Department of Computer Science
University of California, Irvine, CA 92697-3430
{qhan,seguti,nalini}@ics.uci.edu

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 monitoring and network monitoring tools operate at network layer with little awareness of application level object communication requirements. This paper explores the possibility, mechanisms and benefits of filling the gap between object messaging and system monitoring. We introduce the 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 inter object communication properties.

1 The Need for Adaptive Middleware

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, anyplace and anytime. Such distributed applications demand a high degree of flexibility and adaptability in order to deal with (a)changes in application requirements and (b)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 network layer with little awareness of application level object communication requirements.

This paper 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 (a) *system aware messaging* where changes in system status can proactively trigger object level communication adaptation and (b) *application-aware network monitoring* where inter object communication properties trigger optimization of monitoring policies and parameters.

Such an integration brings with it several benefits. Firstly, exposing distributed/global knowledge of system state to the object messaging layer can allow the application level to transparently adapt to the 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. Secondly, 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 co-located 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 object messaging layer. Optimizations such as those described above can help significantly in conserving resources and power at the endpoints while reducing message processing overhead and delays in the network.

2 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 (such as 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 being 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 of them do). Maintaining accurate system state information is expensive since network and system components must be probed frequently and the changes must be 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 fine tuning network monitoring based on application characteristics can help to ensure that most applications receive information at the desired level of quality while minimizing resource consumption. However,

Therefore, in order to serve the 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 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 desirable level of accuracy; the object messaging layer implements messaging adaptations based on current load and connectivity properties (see Figure 1). Similarly, the

messaging middleware provides application meta-data (e.g. application requirements etc) that the system monitoring middleware may use to adapt its monitoring policies and reduce its management overhead.

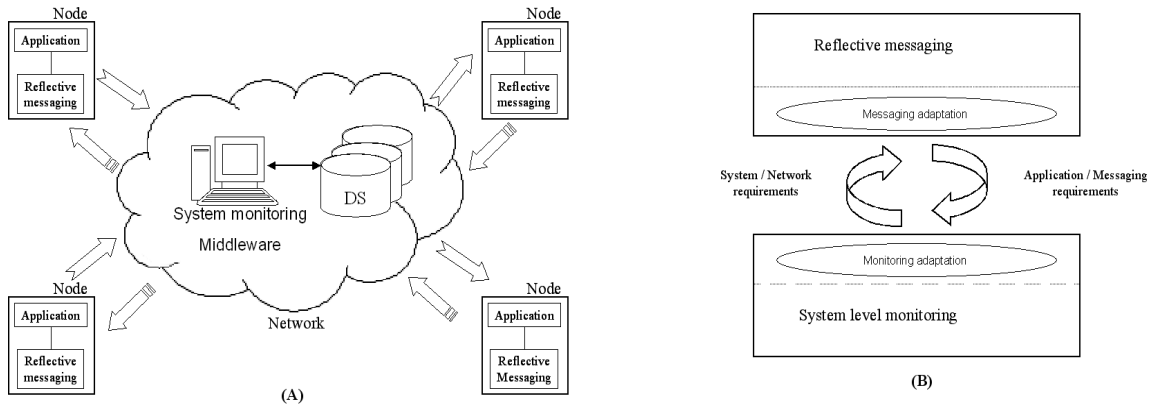


Figure 1: A reflective architecture for bridging system monitoring middleware and reflective object messaging

2.1 The Connection Abstraction

In order to provide customization of messaging services and network monitoring, we develop the connection abstraction, which is capable of: (i) mapping data collected at the network level and translating it into useful information for object messaging, (ii) mapping application communication requirements to network parameters that allow fine tuning of the network monitoring process, and (iii) capturing relevant information about the behavior of the connection.

The connection is a virtual link between two communicating objects distributed in the system and it works as an interaction mechanism between the system monitoring and the object messaging layer. At the system monitoring level, connection endpoints refer to nodes on which objects reside. the connection provides application specific information that helps the system monitoring to specialize and further refine its monitoring process. At the messaging layer level, the connection provides feedback regarding the current system and network status, which is used to customize and fine tune 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 of them with possible different communication requirements. Note that it may be useful to pre-establish a predefined connections between nodes and try to exploit similar communication requirements between objects or to 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 in Figure 2. Formally, we define the connection as a triplet $\nu_1, \nu_2, aparam(u, l)$ 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 in to three phases, according to the connection functionality as follows:

- **Initialization:** A connection may be statically created at startup time, 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

<p>C_{id}: The connection identifier.</p> <p>C_{epist}: The end-point list specifies the pair of nodes/objects (source,target) that share the logical connection.</p> <p>C_{kind}: Connections are classified based on expected duration. Long lived connections are tagged <i>persistent</i>;while short lived connections are tagged <i>transient</i>. Connections may also be <i>mediated</i> if it requires QoS enforcement.</p> <p>C_{req}: The set of communication requirements that the application is interested in maintaining, such as security and timeliness.</p> <p>C_{domain}: The network domain characteristics (if they exist), such as secure.</p> <p>$C_{latency}$: The latency of the connection</p> <p>$C_{loss-rate}$: The average transmission loss rate.</p> <p>C_{load}: The current connection workload in terms of average transmission frequency and size.</p> <p>$Inode_{density}$: The aggregated load from all the connections between two nodes.</p>
--

Figure 2: Connection annotations

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 not longer desired or needed, it is removed to free the monitoring measures at the system and network level.

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

2.2 The System Monitoring Middleware

In this section, we describe the system monitoring middleware architecture(Figure 3) that (a) facilitates the integration of system monitoring and object messaging; and (b) 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 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 and 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. Directory services 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:

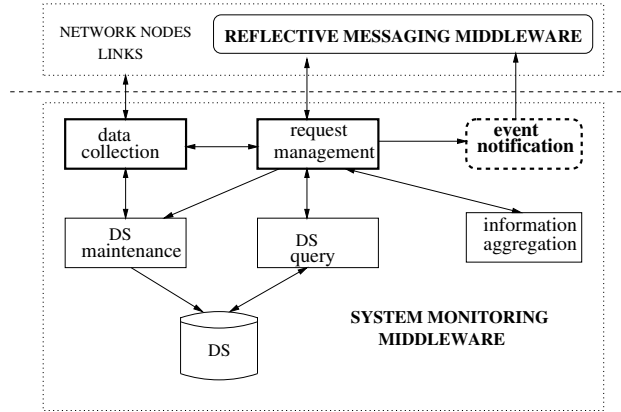


Figure 3: The System Monitoring Middleware Architecture

- 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, to respond to value requests, or to send out notifications when their values are beyond a certain pre-specified threshold. The obtained raw data is sent to *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 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 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.

In Section 3.1, we discuss specific suitable system monitoring policies to facilitate timely response to changes in system conditions.

2.3 Reflective Messaging Middleware

Distributed applications have communication requirements that need to be provided independently 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 (see Figure 4), where each base level object has a meta level object, called *messenger*, which 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

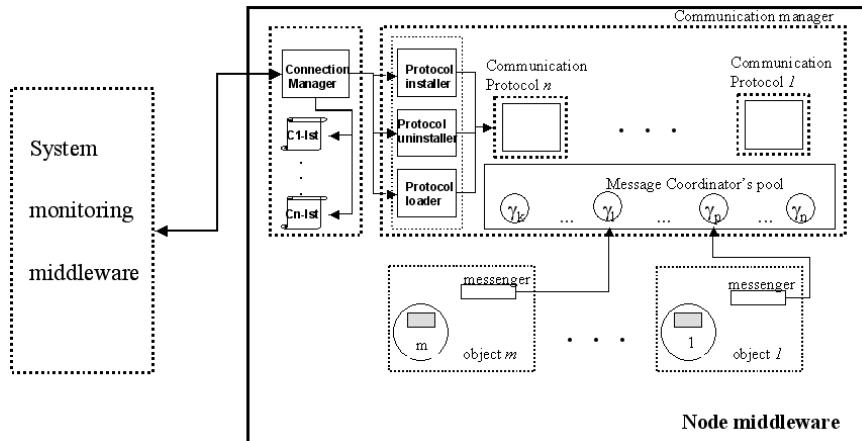


Figure 4: The reflective communication framework model

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 of them 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 assures 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.

3 Application-driven Monitoring and System-aware Messaging

This section describes in detail how the system monitoring middleware and the reflective messaging middleware cooperate to optimize system performance by (a) boosting application satisfaction and (b) 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.

3.1 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, link packet drop rate;
- node parameters: This includes CPU utilization, buffer capacity, disk bandwidth, location (in terms of domain or cell), connectivity, power constraints, throughput.

Keeping track of system conditions in dynamic environments is challenging due to the following reasons. (a) 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. (b) 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. (c) 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 task.

3.1.1 System Monitoring for Dynamic Environments

System monitoring involves two steps: (1) data sampling where system and network level information is obtained from nodes, routers, switches etc. The frequency of sampling may be fixed or varied over time; (2) data updating where the DS is updated based on current sampled values. The effectiveness of system monitoring and the overhead incurred is dependent upon 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 approaches to data representation (Table 1). The simplistic *instantaneous value based approach* [19] 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 approach* [2] can be used to reduce sampling and DS update overhead. It defines a fixed interval which is used to partition the capacity of the collected information into a fixed number (say n) of equal size classes. The classes are represented by corresponding indices $0, 1, 2, \dots, (n - 1)$. 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 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 the sampled information. A throttle based [10] approach uses the average value of samples in previous monitoring window to decide whether a range adjustment is needed or not. A more analytical approach using time series based technique [8] 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 the monitoring and derives an optimal condition to ensure the minimized cost [18].

<i>Approach</i>	<i>Sampling freq.</i>	<i>Data format</i>	<i>Advantages</i>	<i>Disadvantages</i>
instantaneous value based	fixed	single value	easy to implement	high overhead
static interval based	fixed	fixed range	easy to implement	hard to select interval size
dynamic range based	fixed/varying	varying range	decreased overhead	complicated process

Table 1: A comparison of system monitoring approaches

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 [12] 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 fine tune 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 location.

In future networks, fixed host and mobile hosts will co-exist. 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 current network domain is very high, keeping track of individual host mobility is necessary since aggregate host mobility only provides a coarse-level information.

3.1.2 Fine-tuning System Monitoring for Object Messaging

Application-aware system monitoring (i) provides system meta-data (especially connection oriented information) to reflective messaging middleware and (ii) adapts monitoring policies and parameters based on application meta-data and inter object communication properties.

Table 2 describes 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,

<i>Data type</i>	<i>Providing mechanism</i>
network domain characteristics C_{domain}	retrieve directly from the DS
connection latency $C_{latency}$	infer from aggregate link latency
connection packet drop rate $C_{loss-rate}$	infer from aggregate link packet drop rate
connection load C_{load}	infer from aggregate link load
inter-node density $Inode_{density}$	infer from aggregate connection load

Table 2: system meta-data for object messaging

since the DS maintains a mapping between 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 need to be installed or not. If the node moves out of current domain, a notification may be sent to messaging layer if necessary for possible messaging adaptation.

Calculating parameters such as connection latency $C_{latency}$, packet drop rate $C_{loss-rate}$, load C_{load} and inter-node 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., $C = \{p^1, p^2, \dots, p^i, \dots, p^n\}$, $p^i = l^1 \rightarrow l^2 \rightarrow \dots \rightarrow l^j \dots \rightarrow l^{m_i}$. 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, i.e., $p_{latency}^i = \sum_{j=1}^{m_i} l_{latency}^j$. Connection latency is the lowest path latency if $C_{req} = timeliness$; otherwise, it is the average or random path latency.
- $C_{loss-rate}$: The packet drop rate of a path is the maximum link packet drop rate along the path, i.e., $p_{loss-rate}^i = \max(l_{loss-rate}^j)$, $j = 1, \dots, m_i$. 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, i.e., $p_{load}^i = \max(l_{load}^j)$, $j = 1, \dots, m_i$. Connection load is the average path load, i.e., $C_{load} = \text{avg}(p_{load}^i)$, $i = 1, \dots, n$.
- $Inode_{density}$: 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 (see section 3.2 for examples).

At the same time, system monitoring mechanisms can take advantage of application knowledge and inter-object communication properties to fine-tune the monitoring process. Table 3 shows the application meta-data provided by messaging middleware and how this data is used by system monitoring.

<i>Data type</i>	<i>Purpose</i>
connection endpoint list C_{epilst}	infer connection density
connection type C_{kind}	infer stability of system status
communication properties C_{req}	infer dominant communication pattern
triggering conditions for connection latency	send notification when latency changes
triggering conditions for connection packet drop rate	send notification when packet drop rate changes
triggering conditions for connection load	send notification when load changes

Table 3: application meta-data for system monitoring

Three connection attributes (C_{epilst} , C_{kind} , C_{req}) are used for system monitoring middleware to infer inter object communication properties and application workload, which will then be used for customization of monitoring policies and parameters as follows:

- C_{epilst} : 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 a node. Maintaining C_{epilst} 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 $C_{latency}$, $C_{loss-rate}$ and C_{load} for one connection can be re-used 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 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, seamlessly providing 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 tradeoff, 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 [1, 5, 20]. Recent work [11] has addressed the timeliness/accuracy/cost tradeoff 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 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 the other regions. Note that, in this case, latency information should be monitored more closely than the other information.

3.2 Adaptive Messaging

The reflective messaging layer customizes messaging protocols to adapt to changing network and system conditions. This is achieved by fine tuning 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 to 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, latencies vary from 20-200 ms and bandwidth between 0.5-5 Mbps. 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 a 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. e.g. key length is a tolerance parameter for a secure protocol; whereas the acknowledgment type (positive or negative) as well as the sender/receiver window size and the 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 the system monitoring during the three phases of connection management. An initial connection registration message $ConnReg(\nu_1, \nu_2, aparam(u, l))$ is sent to the monitoring layer, where ν_1, ν_2 are the end-point 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, the initial connection latency, drop rate and domain characteristics respectively.

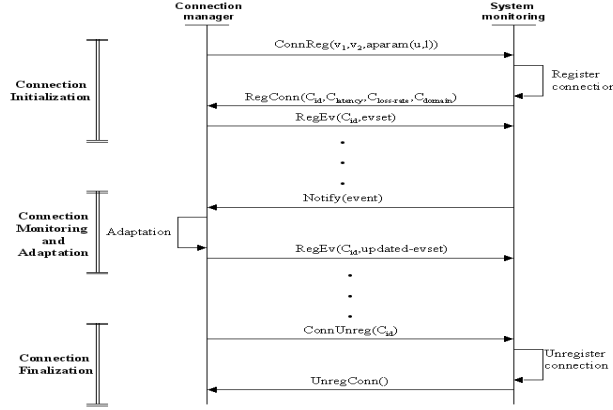


Figure 5: A event diagram depicting exchange of information using the connection abstraction

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(Cid, evset)* to the system monitoring. System monitoring notifies the connection manager when a range violation occurs (*notify(event)*), which may trigger an adaptation procedure. A message *Regev(Cid, updated-evset)* is sent to the system monitoring to update the monitoring parameters (if needed). This process of connection monitoring and adaptation is repeated until tear down, when a message of the form *ConnUnreg(Cid)* 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 that shows how the communication fine tuning 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; else the message is discarded. Given these requirements, the object messaging selects a reliable timely delivery protocol with timeouts, positive acknowledgment and selected retransmissions. After receiving the connection status message *RegConn(Cid, Clatency, Clossrate, Cdomain)*, the connection manager at the node specifies the connection latency and loss-rate as the parameters to be closely monitored. In case of 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 end-point.

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 end-points of a message co-reside in a secure domain; hence 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 is moving out from a secure domain. In this particular case, the system monitoring middleware will notify the connection manager (of both end-points) with a domain change event that triggers the installation of a secure protocol S_1 to guarantee message integrity, secrecy and authenticity at both end-points. 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 re-keying operations need to be executed. Thus, messages may face temporary delays due to re-keying, contributing to further communication overhead.

A solution to this problem will involve a 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.

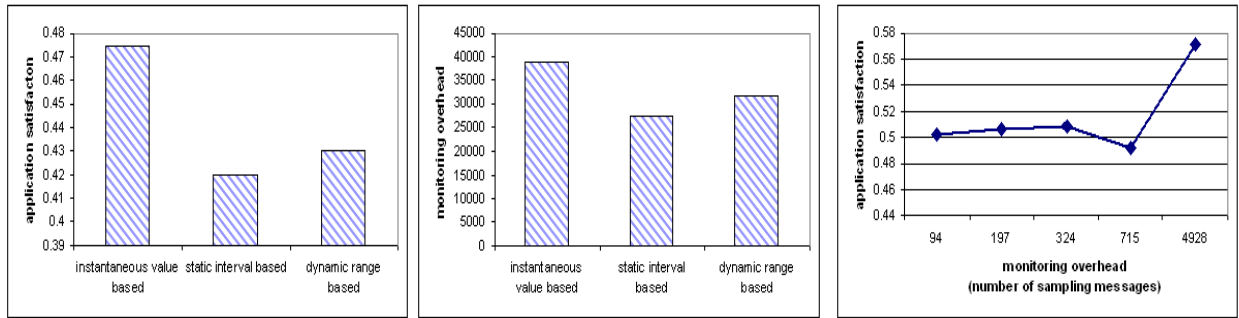
4 Implementation and Performance Issues

In order to evaluate the benefits of application-driven monitoring and system-aware messaging, we explore: (1) the relationship between application satisfaction and monitoring overhead (2) the reflective messaging overhead, and (3) the performance gain due to message adaptation.

The effectiveness of system monitoring is dependent upon two factors: the application satisfaction and the 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 Figure 6A. An analysis of the performance results indicates that the monitoring policy plays a significant role in application satisfaction and the incurred overhead. The dynamic range based policies provide a good balance between application satisfaction and the incurred overhead. Intuitively we would expect the application satisfaction to increase with a rise in sampling frequency (i.e., increasing monitoring overhead); Figure 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 the messaging layer.

We next determine the overhead introduced in the reflective messaging layer by categorizing the different overheads involved in the messaging process (see Table 4). First, we measure the message transmission and reception overheads of the 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 4 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 (see Figure 7). We compare the adaptive and non-adaptive versions of a reliable timely delivery protocol described in section 3.2. The non-adaptive 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



A. application satisfaction comparison B. monitoring overhead comparison C. application satisfaction vs. overhead

Figure 6: System monitoring performance issues

<i>Messaging Overhead</i>	Send (μsec)	Receive (μsec)
Network	181	141
unprocessed message	670	607
Time-sensitive messaging	1891	1942
Secure timely messaging	3133	3280

Table 4: Reflective messaging overheads: message transmission (send) and reception (Receive) overheads displayed by category.

this is capable of modifying connection parameters in order to adapt to the 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 the 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 uninstalls (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.

5 Concluding Remarks

Existing research on system/network resource monitoring and management [17, 21] has focused on algorithms for cost-effective monitoring to provide reasonably accurate system conditions to applications. The Darwin system [4] provides application-specific resource management support by adding resource allocation software into network components without modifying the underlying network architecture. Genesis [13], on the other hand, constructs reflective network architectures to enable re-configurations of existing network services and modifications of network attributes using dynamic plug-ins. In comparison, this paper 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 customization of messaging protocols [6, 7]. In recent years, several group and point-to-point communication systems have been developed to provide dynamic composition of messaging protocols [9, 15, 16, 14]. much of this work has focused on mechanisms to provide dynamic composition, installation

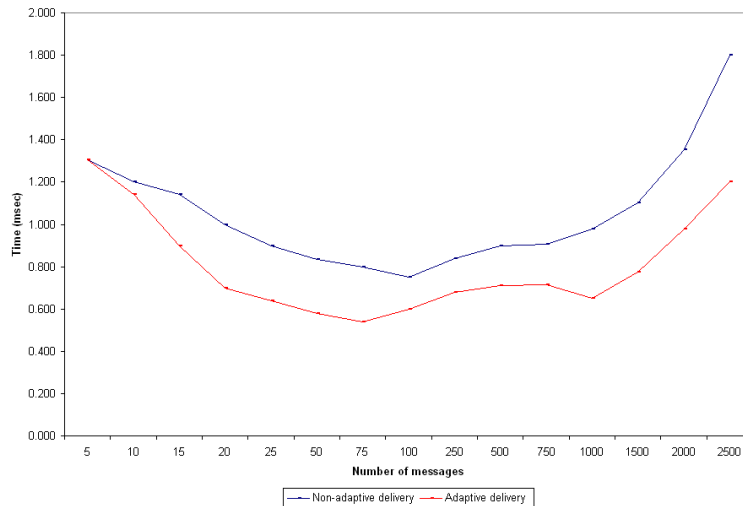


Figure 7: Object messaging performance issues

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 paper, we introduced the connection abstraction as an interaction mechanism between object messaging and system monitoring. We further developed 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: (i) fine-tuned object messaging, and (ii) 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] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying update streams in a soft real-time database system. In *Proceedings of ACM SIGMOD*, 1995.
- [2] G. Apostolopoulos, R. Guerin, S. Kamat, and S. Tripathi. Quality of service based routing: A performance perspective. In *Proceedings of ACM SIGCOMM*, 1998.
- [3] Benoit Garbinato and Rachid Guerraoui. Flexible Protocol Composition in BAST. In *IEEE International Conference on Distributed Computing Systems*, 1998.
- [4] P. Chandra, A. Fisher, C. Kosak, T. S. Ng, P. Steenkiste, E. Takahashi, and H. Zhang. Darwin: Resource management for value-added customizable network service. In *Proceedings of ICNP*, 1998.
- [5] A. Datta and I. Vigiuer. Providing real-time response, state recency and temporal consistency in databases for rapidly changing environments. *Information Systems*, 22(4):171–198, 1997.
- [6] Fabio Costa, Gordon Blair and Geoff Coulson. Experiments with Reflective Middleware. Technical Report MPG-98-11, Lancaster University, 1998.
- [7] Fabio Kon, Manuel Romn, Ping Liu, Jina Mao, Tomonori Yamane, Luiz C Magalhes and Roy Campbell. Monitoring and Security and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM Middleware 2000*, 2000.

- [8] Z. Fu and N. Venkatasubramanian. Adaptive parameter collection in dynamic distributed environments. In *Proceedings of IEEE ICDCS*, 2001.
- [9] S. Gutierrez-Nolasco and N. Venkatasubramanian. A reflective middleware framework for communication in dynamic environments. In *Proceedings of DOA*, 2002.
- [10] Q. Han and N. Venkatasubramanian. Autosec: An integrated middleware framework for dynamic service brokering. *IEEE Distributed Systems Online*, 2(7), 2001.
- [11] Q. Han and N. Venkatasubramanian. Addressing timeliness/accuracy/cost tradeoffs in information collection for dynamic environments. Technical report, University of California-Irvine, 2002.
- [12] Q. Han and N. Venkatasubramanian. Aggregation based information collection for mobile environments. *Journal of High Speed Networks*, 11(3,4), 2002.
- [13] M. E. Kounavis, A. T. Campbell, S. Chou, F. Modoux, J. Vicente, and H. Zhuang. The genesis kernel: A programming system for spawning network architectures. *IEEE JSAC*, 19(3), March 2001.
- [14] Mark Astley, Daniel Sturman and Gul Agha. Customizable Middleware for Distributed Software. In *Communication of the ACM*, 2000.
- [15] Mark Garland Hayden. *The Ensemble System*. PhD thesis, Cornell University, 1998. Department of Computer Science.
- [16] Matti A. Hiltunen, Vijaykumar Immanuel and Richard D. Schlichting. Supporting Customized Failure Models for Distributed Software. In *Distributed System Engineering*, (6):103-111, 1999.
- [17] N. Miller and P. Steenkiste. Collecting network status information for network-aware applications. In *Proceedings of IEEE InfoCom*, 1999.
- [18] C. Olston, B. T. Loo, and J. Widom. Adaptive precision setting for cached approximate values. In *Proceedings of the ACM SIGMOD*, 2001.
- [19] M. Stemm, R. Katz, and S. Seshan. A network measurement architecture for adaptive applications. In *Proceedings of IEEE InfoCom*, 2000.
- [20] R. Zhang, C. Lu, T. F. Abdelzaher, and J. A. Stankovic. Controlware: a middleware architecture for feedback control of software performance. In *Proceedings of ICDCS*, 2002.
- [21] J. Zinky, J. Loyell, and R. Shapiro. Runtime performance modeling and measurement of adaptive distributed object applications. In *Proceedings of DOA*, 2002.