Case Studies of Node-level QoS Adaptation in Cougaar

John Zinky, Richard Shapiro, Sarah Siracuse, Steve Ford, David Wells
BBN Technologies, and Object Services and Consulting
jzinky@bbn.com rshapiro@bbn.com ssiracus@bbn.com ford@objs.com wells@objs.com

Abstract

Multi-agent systems, like other software systems, have a set of abstract resource requirements for proper execution. These requirements, often known as Quality Of Service (QoS), change dynamically and can come into conflict with the availability of physical resources. The ability to adapt QoS requirements to changes in the environment is a key survivability feature in distributed multi-agent systems (MAS). For example, network and CPU resources are heavily stressed in operational environments, and systems that cannot adapt their requirements dynamically will be unable to function effectively. The Cougaar architecture supports successful QoS-adaptation by providing (1) a wide range of adaptation strategies, (2) a reliable characterization of the cost-benefit tradeoff for any given strategy, and (3) a high-level, accurate and timely data model of the dynamic resource constraints and application requirements. With this support available, a society of software agents can pick an effective strategy for adapting to whatever resource-limited situation it finds itself in, and therefore remain operational. This paper presents examples of the use of Cougaar mechanisms and node-level services that improve QoS-adaptation, along with discussions of these various techniques, both runtime and post-mortem, for evaluating the overhead these mechanisms impose. It also presents a detailed description of a suite of test societies, used in our examples, that has proven to be very useful for benchmarking Cougaar and evaluating adaptation options.

1 Introduction

Real-world agent societies must be able to run in resource-constrained environments in which both the QoS requirements and resource availability change constantly. In order to survive and run effectively in these environments, agent societies need the ability to adapt dynamically, that is, to change their behavior to overcome the constraints imposed by the environment. No single adaptation strategy is effective in all situations. But if a collection of strategies is available, each of which works in a given context, then the system can adapt and survive as long as it can make a reasonable choice from the set of strategies. In other words, to adapt effectively, an agent society needs to provide support for the following kinds of capabilities:

- Interchangeable adaptation strategies: Multiple adaptation strategies must be available along with mechanisms to switch among them.
- Cost/benefit characterization: Each adaptation strategy must be characterized in terms of its QoS properties and its resource consumption.
- System metrics: The status of the environment must be observed, including the society's QoS requirements and the system's available resources.

The infrastructure of the Cougaar architecture supports these capabilities at multiple timescales. QoS-adaptive features can be mixed and matched to make Cougaar societies that meet a wide spectrum of QoS requirements, from fast simple societies that run on a single host to large robust societies that can be distributed across a potentially hostile environment.

At configuration time, QoS-adaptive features can be added to or removed from a society so that they impose a minimal cost for any given environment. Cougaar's pluggable component model can be used to configure systems with the right set of adaptation and metric features for the QoS environment in which they will run.

At runtime, QoS-adaptive features can change their behavior over time or in the context of specific agents or sets of agents. Cougaar offers several node-level services to help create dynamic QoS-adaptations [1]. The Metrics Service provides timely aggregated data, useful for both measuring the current QoS and also for measuring the cost and effectiveness of an adaptation. The thread control services provide a means of adapting to changes in CPU-related QoS. Finally, the message transport service's pluggable protocols provide a means of adapting to network and security related QoS.

This paper will describe a series of specific examples that will show how QoS-adaptive features are used to improve the performance and scalability of a simple Cougaar test society. We also present some basic performance metrics for Cougaar 11.2. We believe these examples will show that QoS adaptation is both possible and effective in Cougaar, and that Cougaar's supporting services make the process of adding QoS adaptation clean and straightforward.

Section 2 introduces the simple Ping society we use throughout the paper as the testbed society. The next three sections (3-5), describe the use of the Ping society and characterize the overhead of various QoS adaptive

features. Three kinds of experiments are described in detail. In the first (section 3), the capacity of a physical resource is characterized, as a benchmark. In the second (section 4), the relationship between a society's configuration and its consumption of resources is characterized. In the third (section 5), the two preceding characterizations are used to identify performance bottlenecks in different society configurations. The next three sections (6-8) show how to add dynamic QoS-adaptation to various subsystems. Section 6 shows how to add instrumentation. Section 7 shows how to switch message transport protocols dynamically. Section 8 shows how and when to add message compression, particularly in the context of a large logistics society. [3]

2 Cougaar Ping Societies

2.1 What is a Cougaar Ping Society?

Since the deployment of 500-to-1000-agent logistics societies [4], scalability has become a major factor in the performance of Cougaar. In order to isolate various combinations of components and to achieve an easy way to observe the associated overhead and performance improvements, we developed a simple society we call "Ping', along with Ruby-based society configuration builders using ACME [3]. Ping includes a large array of configuration options and post-processing tools, and grew to be a very efficient mechanism for exposing caveats in Cougaar configuration overhead, relevant bugs, and the effectiveness of loaded QoS components.

Ping consists of Blackboard Relays, which update Ping object information. On the source-side, two plugins manage this: one that adds the initial Ping object (PingAdderPlugin), and another (PingTimerPlugin), that periodically wakes up to update and send the Ping object, and log a 'CougaarEvent' with basic aggregate statistics. Parameters can be added at configuration time, including ping start time, delay between pings, and the size of ping messages. Construction of a society is done with ACME XML preprocessing tools, which easily configure a Ping society and then run it. This streamlined process enhances Ping's flexibility as a testing tool to analyze configuration overhead and performance.

2.2 Flavors of Ping

Ping plugins can be configured to run in various society configurations. Each layout configuration can expose various kinds of overheads and bottlenecks. In our characterization of Cougaar, we use two basic configurations, both of which address differing performance issues.

2.2.1 Single Ping Society

A single Ping society, the one most commonly used in performance analysis, contains one ping pair that is run continuously, i.e., with no delay between pings. The society has one source and one sink agent that can be configured on either one or two Cougaar nodes. For the single ping loop, all processing associated with the ping is executed serially. When a QoS component is added, any overhead will add latency to the ping loop. The performance impact can readily be measured, simply by measuring the rate of pings per second. The count of pings is logged and the change in the count over fixed time intervals is used to calculate the rate. The single ping society is a good mechanism for baseline and configuration overhead tests. Subsequent tests, when compared with the baseline, expose the latencies (increases in round-trip time) associated when adding additional services and survivability components. We use this simple configuration model to isolate and expose many facets of memory usage and message overhead, as illustrated by following case studies.

2.2.2 Multiple Ping Society

Multiple, or distributed Ping, runs multiple ping loops continuously and in parallel, between either two Cougaar nodes running on one or two hosts, or multiple unique nodes on many hosts. Each ping loop runs independently and competes with other ping loops for processing and network resources. The multiple Ping loop executions naturally queue up to use the available network and CPU resources, exposing crucial resource bottlenecks.

The multiple ping society exposes bottlenecks in the following manner. Once a ping loop gets its chance to use a critical resource, the rest of its loop processing can be done in parallel with other ping loops, which can use the critical resource until the first ping loop is again queued to use the critical resource. The overall ping rate will increase as more pingers are added, until the bottleneck becomes saturated. Adding more ping loops beyond this saturation point should maintain the same overall ping rate, but the ping rate for each ping loop will decline, which is related to the longer queue wait for the bottlenecked resource. Thus, the aggregate ping rate for all ping pairs is limited by the processing speed of the bottleneck resource. In our examples, we calculate this overall rate by offline post-processing tools, which aggregate and analyze Ping events in the society log files. By changing how the ping pairs are configured over resources, bottlenecks can be exposed. Also, in many of our examples we use a "corset" topology of source and sink ping pairs, residing on two respective nodes. This multiple Ping society is useful for detecting overhead associated with scaling up the society, such as linear searches through lists of agents or congestion waiting to use a resource.

2.3 Uses of Ping

We can use the Ping societies to characterize the overhead associated with society component configuration and QoS services, but environmental

factors must also be taken into account in order to gather meaningful results. To address these factors, an analysis of system benchmarks is necessary when considering performance comparisons across multiple platforms and networks. The next three sections explain how Cougaar mitigates three of these important resource issues: 1) CPU capacity of the host running the experiment, 2) Assessment of cost benefit for a given QoS-adaptation in different situations, 3) Parallelizing the society to run on a multiprocessor.

3 Benchmark Ping Society

This section describes how to characterize the operating environment of the society. Cougaar societies can run on multiple types of hosts, each of which can vary in terms of hardware architecture, clock rate, number of processors, operating system, and Java virtual machine. When a Cougaar node starts up, a quick CPU benchmark of the host is run and then published in the Metrics Service. This benchmark (measured in Millions of Java instructions per second, or MJIPS) is strongly correlated to the performance of the society under different host configurations. Off-line analysis is used to predict the performance of a society, and is based on the MJIPS benchmark.

We have found that Linux running on Intel Pentium-4 processors have performance characteristics that differ from those of a Sun or Mac OS-X platform. While experiments could be run on a single family of processors, the results would be hard to apply to other families or even to later generations of operating systems or Java virtual machines.

A CPU benchmark is a general method of aggregating the effects of different host configurations into a single capacity metric. The standard Cougaar Metrics Service runs a benchmark on the host when the node is started. The performance of the society under different host configurations can be correlated to the benchmark of the corresponding host configuration. Note that the Cougaar benchmark measures the capacity of a single processor on the host; this allows for easier modeling of multiprocessor hosts, as we will see in Section 5.

To illustrate the usefulness of the CPU benchmarks we conducted a simple experiment running a minimalist Ping society on a variety of host configurations. The minimalist Ping society has a single ping pair between two agents on the same node and host. Also, many of the basic Cougaar services were replaced with trivial, *single-node* implementations that do not support runtime observation or control over the services. While these minimalist services do not support QoS-adaptation, they can be used as a base line for calculating the overhead for services, as shown in Section 4. For this paper, the minimalist society uses trivial implementations of the Message Transport, Metrics, and Thread services; it also does not load the Servlet, Community, or Yellow Pages services. This Ping society is designed to run as fast as possible, but it uses only one thread. The single-thread restriction does not allow parallelism and can be directly compared against the single CPU MJIPS benchmark.

Figure 1 shows the results of running the minimalist Ping society on a variety of host configurations. Linear regression is used to fit the performance of the ping society measured in pings per second to the benchmark capacity of the host measured MJIPS. The fitted value for the slope of 5.2 implies that a single ping uses 0.19-million Java instructions. Notice that this fit is accurate for many host types, some of which include multiprocessor configurations.

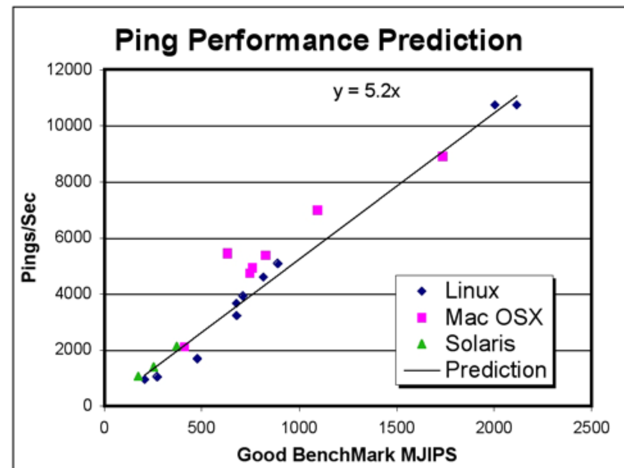


Figure 1: Comparison of *Good* MJIPS Benchmark with a Minimal-Ping Society

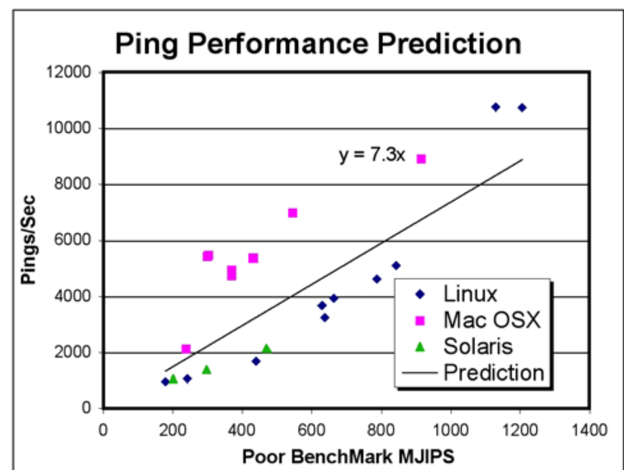


Figure 2: Comparison of *Poor* MJIPS Benchmark with a Minimal-Ping Society

But benchmarks must have an instruction mix that is characteristic of the society if the benchmark is going to accurately predict the performance of the society. Figure 2 shows the results using an earlier benchmark. Notice that each type of host has its own trend line and that the

overall fit is not accurate for any one type of host. This shows the danger of using a benchmark for characterizing host characteristics. The benchmark must evolve with the application, and/or multiple benchmarks must be used.

4 Relative Overhead of QoS-Adaptation

This example shows how different implementation of node-level services can have different QoS properties and can consume different amounts of resources. Depending on the expected environment, the appropriate implementation can be chosen at society configuration time.

Each node-level service performs a function, but associated with that service is an overhead. The benefit and overhead of an individual service is dependent upon the overall configuration of a Cougaar society. For example, a service in one configuration could offer little benefit and have a high overhead relative to other components. This service is a candidate for performance improvement, or even removal from the configuration. Conversely, in another configuration, the same service may have a high benefit, with only a small overhead relative to other components, making the service critical to society. Node-level QoS-adaptation services tend to have the following overhead behavior: high benefit in a certain situation, but not in others. Local vs. distributed configurations are classic examples of when QoS-adaptation service is needed, and when it becomes a performance liability, it can be removed. A major feature of Cougaar is that QoS-adaptation can be removed for a local configuration and added back for distributed configurations. (Processing in a local situation does not usually need QoS-adaptation because the network resources do not change.)

In local configurations, the overhead for a QoS-adaptive service is relatively high. Table 1 illustrates the differences in societal configuration overhead. The first row shows the relative cost of a standard node-level services. The minimal Ping society described in section 2.2.2 uses the rudimentary Metric and Thread services, When these services are removed the performance increases by a factor of three. If the Metrics Service and thread control are unnecessary, these services can and should be removed at configuration time.

In distributed configurations, message serialization dominates overall society performance. QoS-adaptation services become necessary to react to changes in the networked resources, such as device failure, competing network traffic, and security attacks. In addition to the value of such OoS services, their relative overhead is small. Table 1 presents the overhead associated with message serialization in the context of performance. Row 2 of Table 1 shows a performance decline associated with just serialization of messages. These results reflect an additional test plugin in our minimal Ping society, which forced the serialization of messages, even when sent locally, resulting in over a third decrease in performance. Row 3 exposes the combined overhead of using RMI and serialization, when the source and sink agents were separated onto two nodes running on the same host. Notice that there is almost a factor of 10 drop in performance when sending pings between node processes. Despite this, the relative performance increase of removing the metric and thread services is a trivial 21%. Since the benefit of these services is high and the relative overhead is low, this simple test suggests that these QoS-adaptive services should be installed in distributed configurations.

Table 1: Relative Overhead of Node-Level Services (Single Ping, 2500 MJIPS Processor)

Configuration			Performance (Pings/second)		
Serial	RMI	2 Hosts	Standard	Minimal	%
			1061	3229	304%
X			306	466	152%
X	X		139	165	121%
X	X	X	179	210	117%

The last row of Table 1 illustrates how the performance actually increases slightly when the nodes are run on two hosts connected by a high-speed network. The increase is due to the fact that some of the society processing can be done in parallel with the ping loop. Parallel execution of the ping society will be discussed in the next section.

5 Performance of Parallel Ping Society

In addition to analysis of the benchmark society, performance characterization must also be taken into account to show the relationship of a society's configuration and the underlying resources. In this example, the society is configured so that its processing can be done in parallel. If the underlying resources can execute in parallel, such as with a multi-processor host, then the overall performance increases with an increase in parallelism. But too much parallelism can actually decrease performance due to congestion. The society will not scale if too much overhead is associated with deciding what work to do next.

The multiple Ping society has multiple loops that can be run in parallel if multiple processors are available. In contrast, in the single Ping society, which is used in the experiments discussed in the previous two sections, all the work must be done serially, and therefore it cannot benefit from multiple processors.

For a single processor host, the additional load offered by the multiple pingers just slows each loop, but the overall rate should stay the same. The rates for each ping

pair should be the same; if they are not, the scheduling infrastructure isn't *fair*. For instance, thread control options using the Cougaar Thread Service's thread lanes can give one ping pair preference over others. [1]

For Multiprocessor hosts, some work can be done in parallel, with each processor taking the work for a ping pair. For one ping-pair there is only enough work to keep one processor working. As more ping-pairs are added, processors will process their work independently and in parallel. The overall ping rate saturates when the number of processors is less than the number of ping-pairs. Again the individual ping rate will go down but the overall saturated rate will remain constant. The overall ping rate circulates approximately at the base ping rate multiplied by the number of processors.

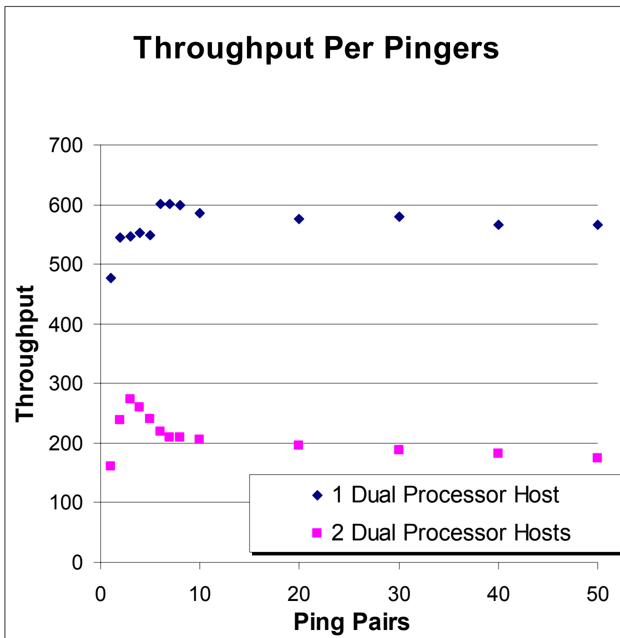


Figure 3: Parallel Execution of Multi-ping Society

For distributed pingers, multiple hosts can work in parallel. As noted in the previous section concerning overhead, some work done by the processors is non-Ping related. Other work includes processing the sender of one ping loop while the receiver is processing the sink of another ping loop. (The network transmission can also be done in parallel, but that is not germane to this example.)

Figure 3 shows the multi-ping society running two configurations. The hosts have dual processors that run at 1250 MJIPS, so the baseline results are about half those in Table 1. For the single-host case, the performance saturates at about two ping pairs. For the two-host case, there are four processors working, so the performance does not saturate until four ping pairs. Also, as the society gets larger, the traffic congests and performance actually goes down. Poorer performance under congestion is an undesirable feature and will be addressed in future releases of Cougaar, as a result of this characterization.

6 Runtime System Metrics

This example shows Cougaar runtime support for metrics of QoS requirements and resource constraints. The Cougaar Metrics Service supports the collection of and access to system performance metrics at runtime. Multiple layers are used by the Metrics Service to collect, process, and disseminate the statistics. These layers and their component can be added or removed to match the needs of the society. Servlets and other clients can subscribe to the metrics and change their behavior based on the values. Node-level components can be added to measure the agent load and publish the results into the Metrics Service. We will show how the Metrics Service itself can be configured to give metrics of varying quality according to the requirements of the society.

The **Metrics Service** integrates measurements from multiple sources and makes high-level interpretations of system metrics available in real time. The Metrics Service keeps track of the credibility of the data it receives and uses this information when merging measurements from multiple sources. The Metrics Service also has a simple model of the Cougaar society and can infer high-level metrics by applying formulas and by assessing the relationship between objects. For example, the effective CPU capacity available to an agent can be calculated by integrating low-level metrics. The Metrics Service looks up the characteristics of the agent's host: the number of processors, their MJIPS, and the hosts load average. These low-level metrics can come from a system/network management system, or be sent from remote host via a "gossip" protocol. [1]. The resulting high-level metric has both its value and its credibility calculated from its low-level metrics. The Metrics Service is available for reading and writing metrics from any component in the Cougaar node. This allows the exchange of system metrics between components that have no direct interactions.

Metric Servlets are clients of the Metrics Service that format the metrics for viewing via a web page. About a half dozen types of Metrics Service Servlets can be loaded into the society. Figure 4 is a screen shot of the Agent Load Servlet, which shows the load that agents and services are putting on different node resources. For example, the "Agent A" row shows that Agent A is consuming 1057 MJIPS of CPU and is generating and receiving 1012 messages a second. Note that this is the same single-node ping society as in the standard case of Table 1. Also, note that CPU load is given for the Metrics Service and the Message Transport Service (MTS).

Sensors are inserted into the workflow of the base Cougaar infrastructure. The Thread Service and the Message Transport Service have specific interfaces for adding components that can monitor their internal operation. The Thread Service has a Thread Listener Service, which calls back its clients whenever threads

change their run status. The MTS allows plugins to be attached that can intercept messages at multiple places in the processing chain. Gathering statistics is the simplest use of these services. For example, the agent CPU load can accumulate the time that threads are running by measuring the time between start-running and end-running callbacks. Also, messages to and from agents can be counted by plugins attached to the MTS. We encourage accumulating only counts in the monitoring hooks, because these interceptions are in the critical path and should have minimal overhead. The raw counts are published in services that can quickly take a snapshot of the statistics. Processing of the statistics is done outside of the sensors.

Differentiators are used to sample the sensor services periodically to get a snapshot of the accumulated counts. The average rate is calculated by subtracting past snapshots from the current snapshot and dividing by the time between snapshots. Keeping a series of decaying snapshots supports multiple averaging intervals. The processing of the counts is done outside the sensor loop, i.e., in parallel with the real workflow. This separation of the collection and processing of statistics reduces the latency for completing the workflow and allows the processing to be done on multiprocessor machines and/or at lower priority. The calculated rates are published in the Metrics Service.

The Metrics Service and its sensors provide the raw data that is needed to make QoS-adaptation decisions. These metrics are available in real time, through a variety of mechanisms. Local components can subscribe to the Metric Reader Service, while operators and post processing tools can access metrics via web pages and society log files. Metrics are also shared between nodes, by piggybacking metrics on messages using the gossip protocol. Thus, adaptive decision can be based on measurements that were collected remotely and transferred to the local node.

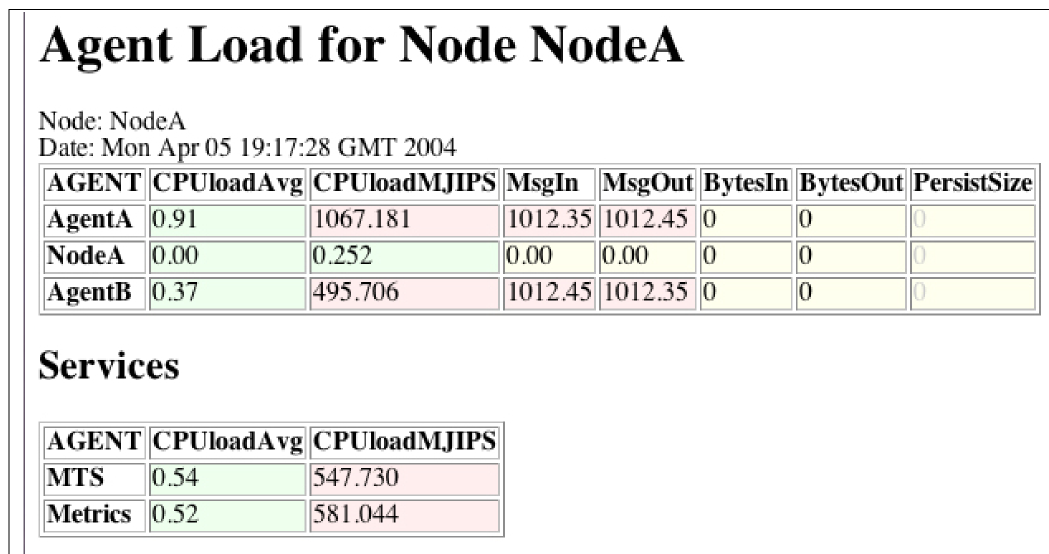
7 Link Protocol Selection

This example illustrates a QoS adaptation that can change behavior at run time. The system in the example dynamically selects which transport protocols to use in a variety of failure scenarios. This adaptation is the result of the following characteristics of the Cougaar and UltraLog messaging systems:

- The ability to send inter-agent messages over any of an extensible collection of communications protocols.
- The ability to collect statistics on messaging behavior to aid in the selection of the protocol to use.
- The ability to dynamically select the best protocol for a given message based on current and past messaging behavior and application-declared semantics of the message.

These adaptations are loosely encompassed in Ultralog's Adaptive Robustness Defense Thread's *MsgLog*, a defense concerned with maintaining message integrity under compromised or significantly degraded network resources. [3]

MsgLog starts with the ability to send messages via any of an extensible set of communications protocols, the current set being RMI, SSL, Loopback, SMTP, NNTP, UDP, TCP, Email, and CORBA. These protocols have been adopted because they are *industrial strength* and provide a wide range of capabilities and failure modes.



Agent Load for Node NodeA

Node: NodeA
Date: Mon Apr 05 19:17:28 GMT 2004

AGENT	CPUloadAvg	CPUloadMJIPS	MsgIn	MsgOut	BytesIn	BytesOut	PersistSize
AgentA	0.91	1067.181	1012.35	1012.45	0	0	0
NodeA	0.00	0.252	0.00	0.00	0	0	0
AgentB	0.37	495.706	1012.45	1012.35	0	0	0

Services

AGENT	CPUloadAvg	CPUloadMJIPS
MTS	0.54	547.730
Metrics	0.52	581.044

Figure 4: Agent Load Servlet (Single Ping, Single Node, 2500 MJIPS processor)

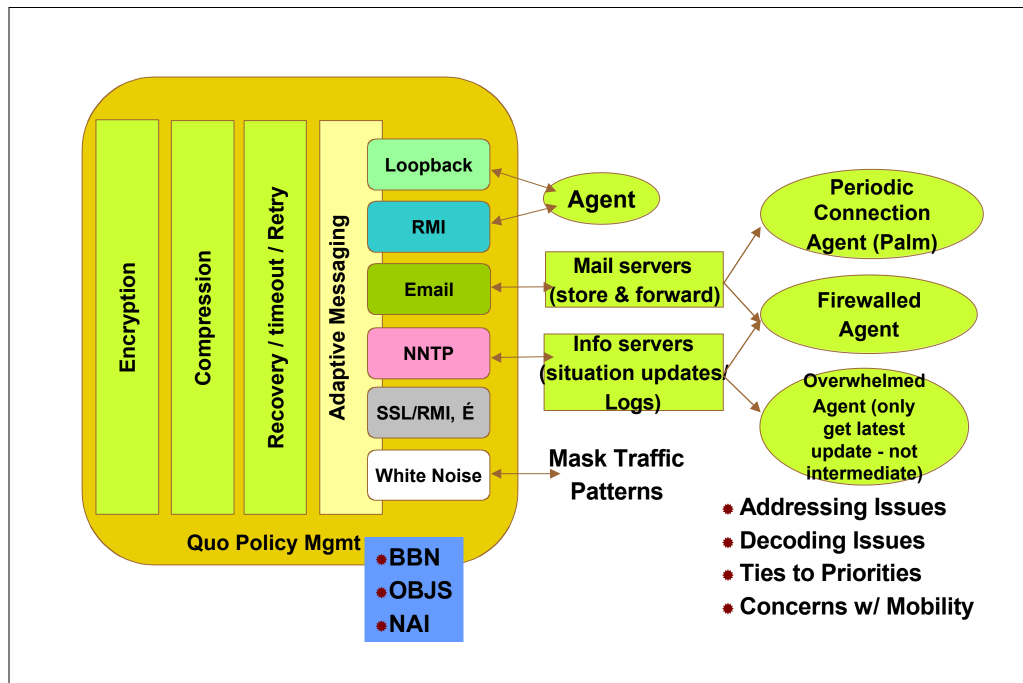


Figure 5: Types of Link Protocols and Their Uses

The point in having multiple protocols available is that each protocol is characterized by different semantic, performance, and failure properties. However, they are not directly usable for intra-agent messaging in the Cougaar software because their semantics do not necessarily match the desired delivery semantics for any message in society. To address this, each protocol is wrapped to bring it up to some minimal level of semantics required for any kind of message. This includes implementing such capabilities as message IDs, externalization/internalization of message content, address lookup, and the ability to access encryption routines. A protocol may also be wrapped to provide optional delivery acknowledgement, guaranteed delivery (with infinite retries if necessary), message ordering, broadcast/multicast, etc.

The protocol selected for any given message is determined adaptively, based on a number of factors. Every message handled by MsgLog has some semantics associated with it, for example, whether a response is required, whether it is a point-cast or a multi-cast, etc. Standard Cougaar delivery semantics (i.e., RMI-like) are assumed by default if not explicitly specified. These semantics must be maintained regardless of which transport protocol is used, but this still allows a choice among several of the wrapped protocols. Protocol selection is based on two primary factors. Under normal network conditions, there is a natural performance-based ordering among the acceptable wrapped protocols for any given delivery semantics. For example, normal point-to-point Cougaar inter-agent messages require delivery receipt and guaranteed delivery, so under normal circumstances they are most efficiently sent using RMI, whereas heartbeat health status messages do not require receipt or guaranteed delivery, can be discarded if too old, and are more efficiently delivered using UDP. Similar statements about protocol desirability can be made about messages being delivered to agents known to be moving or to groups of agents, and about delivery in the face of known communications problems such as frequent partitioning or low bandwidth. Future applications of Cougaar infrastructure are encouraged to supply delivery requirement semantics; at present, semantic relaxation is used only for a few important classes of messages (heartbeats, pings, white-pages messages) whose semantics have been identified.

To choose among the acceptable protocols, MsgLog gathers network connectivity and bandwidth-status information from lower-level sensors and collects delivery-status statistics on messages it has attempted to deliver. Statistics are maintained locally, so each node has a comprehensive picture of messaging from its own perspective. Summaries of this information are lazily passed (using MTS's gossip facility) to the MsgLog component in the Cougaar Management Agent, where a society-wide approximation of network status is constructed. This information is used to notify a higher-level Adaptive Control Defense Coordinator [5] about problems that might involve corrective responses (either requiring or inhibiting) other than messaging. Assuming that no problems have been detected, MsgLog will always attempt the "preferred" protocol first, and will switch to

alternate protocols if the current protocol fails. Failure can take two forms. The first is when the protocol itself returns an exception. An example of this is if RMI cannot obtain an end-to-end connection between source and destination. The second kind of failure occurs when messages are taking an unacceptable amount of time to deliver. While the underlying protocol may still be trying to deliver a message, MsgLog will try alternates when the elapsed time reaches an adjustable multiple (by default two) of the expected-delivery time given current network statistics. If a non-preferred protocol is selected, it will be used for a while and then the preferred protocol will be again selected as a test for whether the original problem has resolved itself. The selection of these protocols is important to mitigating message tracking and integrity because protocols differ significantly in performance, as noted in the comparisons in Table 2. Performance costs become crucial when choosing a protocol for a society when running in bandwidth-stressed networks.

Table 2:Performance of Link Protocols with the Standard Ping Society

Link Protocol	Pings/second	%
RMI	196	100%
SSL/RMI	166	85%
CORBA	184	94%

8 Compression

Compression of messages is an example of trading off CPU resources for network resources. When the network bandwidth is low, for example from heavy traffic or from using the SSL protocol, CPU resources can be used to compress the size of the messages to reduce the amount of data that is being sent. In a sufficiently low-bandwith environment, a compressed message will be delivered sooner than an uncompressed message, even accounting for the time it takes to compress the message. But in the case of high network bandwidth, compressing the message will make the message be delivered later than an uncompressed message, because compressing takes longer than transmitting the raw message.

Message compression is added to a society using an MTS plugin. The plugin inserts functionality into three places in the message processing flow. Compression itself is done by adding an *OutputStream* to Message serialization. A series of *OutputStream* filters is allowed to transform the message. For example, one plugin can add an *OutputStream* to count the bytes in the raw message, while another encrypts the message. The order in the serialization pipeline is important. If the byte counting is put last instead of first, it will count the length of the message being sent on the wire instead of the length of the raw message. Also, compression must come before encryption, because an encrypted stream has no patterns that can be compressed.

Decompression is likewise added as an *InputStream* in the de-serialization pipeline. The order of the filters is sent with the message attributes, so that de-serialization can be performed in the proper order at the receiver. The *InputStream* filters are in reverse order of the *OutputStream*, e.g., decrypting before uncompressing.

The choice of whether to compress can be made dynamically at run time, if the capacities of the network and the hosts are known. The Compression Plugin adds a check just before the message is sent to the LinkProtocol to determine whether it is worth compressing the message. If it is, LinkProtocol adds a message attribute to the message indicating that the *CompressingOutputStream* should be added when the message is serialized.

The criteria for choosing whether to compress depends on several metrics that need to be available at the time the decision is being made. Figure 6 show message throughput curves for not compressing and compressing a 10,000 byte message. In the no-compression case, the message rate is limited by the bandwidth of the network path for low bandwidth situations, but forms a plateau at the message serialization rate for high bandwidth paths. In the compression case, more compressed messages can be transmitted in the same bandwidth because the size of the message has been reduced as measured by the compression ratio. But the plateau is much lower for compression because the CPU must both serialize and compress the message.

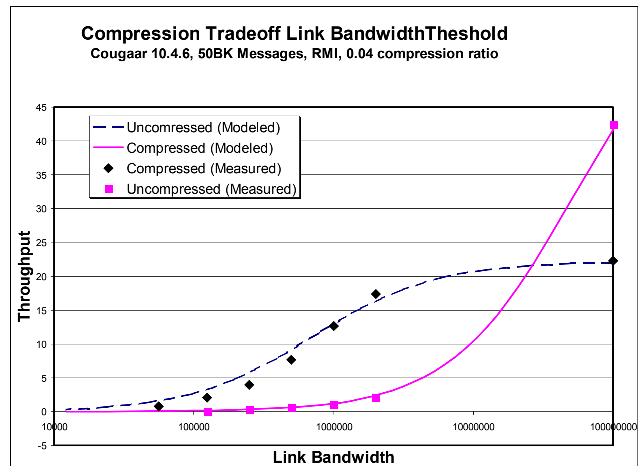


Figure 6: Compression vs. No-Compression

The threshold that determines when compression is no longer useful occurs where the two curves cross. This point is where the compression rate is equal to the network bandwidth. Notice that the threshold is independent of the message size and compression ratio, but rather depends on the capacity of the CPU and the instruction overhead of the compressing messages. Below the threshold compression should be used, and above the threshold compression should not be used.

A consequence of making the wrong decision is that compression could make performance worse. Without metrics for the network bandwidth and CPU capacity a rule of thumb can be used to select when to use compression, such as "*Always compress over a WAN, but never compress over a LAN*."

The Compression Plugin goes beyond a simple rule of thumb and uses the Metrics Service to obtain metrics for inter-node bandwidth and effective capacity for the local and remote hosts. Depending on the credibility of the metrics, different selection criteria are used. For example, if the network bandwidth metric indicates a low-bandwidth path, but the remote-CPU-capacity metric has low credibility, then the threshold for when to switch may be lowered. In practical terms, after the first message is sent to the destination, the remote CPU capacity metric will have high creditability because the CPU measurement will be carried back via Gossip on the delivery confirmation. So if the remote CPU is uncertain and the network bandwidth is close to the threshold, the first message is sent uncompressed, and if the remote CPU capacity is high enough, the second message is compressed.

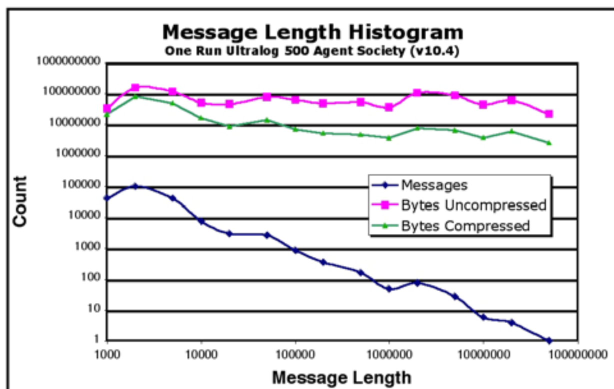


Figure 7: Message Length Distribution for Ultralog 500 Agent Society

The benefit of compressing messages can be great if the compression ratio is high. Cougaar messages contain blackboard objects to be copied to the remote agents' blackboards. Java serialization of these objects contains many class-name strings that are easily compressed. Figure 7 shows the frequency of messages by sizes for the Ultralog logistics society. The top two lines show that small messages tend to be compressed at 3 to 1, while large messages are compressed at 10 to 1.

Note that Figure 7 also shows the message-length distribution for the logistics society. The society sends some extremely large messages, e.g., one message has 28M bytes. Message lengths do not follow the classic Poisson distribution because of the heavy tail of large messages. Another interesting characteristic is that the large number of small messages amounts to a number of bytes approximately equal to a few large messages. This pattern is similar to Internet web traffic, which is considered to have "self-similar" traffic distribution [6]. Finally, there is no way to tell the size of the message before it is serialized. All of these factors make it difficult to try to predict how long it will take to transmit a message.

9 Conclusions

The examples described in this paper demonstrate that the Cougaar infrastructure can be used to add effective QoS adaptation in a modular and measurable way. QoS-related features can be added or removed at configuration time, as appropriate, and when present can react dynamically and appropriately for the runtime QoS context when the society is operating in a hostile or resource-stressed environment. The costs of any given adaptation strategy can be measured accurately and balanced against the survival benefits it adds.

10 Acknowledgements

This research was funded in part under DARPA contract #MDA972-01-C-0025 and is Approved for Public Release, Distribution Unlimited

11 References

[1] BBN Technologies, *Cougaar Architecture Document V10.0*, 20 February 2003, http://www.cougaar.org/

[2] UltraLog Project Site, http://ultralog.net/

[3] Cougaar Open Source Development Site, http://cougaar.org/

[4] Dana Moore, Aaron Helsinger, David Wells, "Deconfliction in Ultra-large MAS: Issues and a Potential Architecture" Open Cougaar Conference, New York, 2004.

[5] David Wells, Paul Pazandak, Marian Nodine, Anthony Cassandra, "Adaptive Defense Coordination", submitted to IEEE Symposium on Multi-Agent Security and Survivability, 2004.

[6] Hernandes-Campos *et al.*, "Variable Heavy Tailed Durations in Internet Traffic, Part 1: Understanding Heavy Tails", *10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'02)* October 11 - 16, 2002