# Performance Comparison of CORBA and Message Queuing under Diverse Network Constraints

**T. Andrew Au**
Defence Science and Technology Organisation, Australia.

DSTO C3 Research Centre, Fernhill Park,
Department of Defence, Canberra,
ACT 2600, Australia.
Email: andrew.au@dsto.defence.gov.au

## Abstract

Middleware provides standardised mechanisms that can be used to simplify the construction of reliable and flexible distributed applications over a network. Applying middleware to C3I problems can provide simple integration of disparate information systems in the military environment. Often, various middleware technologies are used as the communication infrastructure and as a practical ease to the network-programming problem. CORBA and message queuing are undoubtedly two major middleware paradigms that are fundamentally distinct from one another. In order to exploit middleware in the military environment, we need to understand the system characteristics of these middleware technologies A robust middleware infrastructure is a necessary underpinning for effective distributed collaboration across the range of stringent military networks. This paper presents a performance comparison between these two technologies, under typical test environments. We attempt to identify problems that arise from different network conditions in the interaction between client and server. Our experimental results provide guidance in assessing the feasibility of using different middleware paradigms in various military computing environments.

**Keywords**: Middleware, CORBA, Message Queuing, Latency, Response Time, Throughput.

## 1. Introduction

The Experimental Command, Control, Communications, Intelligence, Technology Environment (EXC3ITE) is an Australian Defence Project, to develop and leverage the use of distributed object middleware in the Defence Information Environment. Distributed object middleware is an emerging technology leading from current Internet client-server based systems towards next generation Internet multiple component peer-to-peer networks. The benefits of such middleware architectures include evolvability, reuse, scalability, and reliability, thereby offering significant advantages in relation to our multi-billion dollar information infrastructure. Military communications and information systems are often characterised by the development of expensive, purpose-built and non-interoperable systems. The increasing frequency of joint military operations makes interoperability of these systems within and across Services essential. As investment focuses on interoperable information services and components, the legacy of costly, stove-piped applications will diminish and the vision of scalable information management will be realised.

The systems architecture of EXC3ITE is founded on the concept of using components to encapsulate intellectual property, at a level of granularity that allows ready sharing and re-use. With a suitable set of components, C3I (command, control, communications, intelligence) applications can be constructed rapidly and, where appropriate, by end users. Information services (both remote and local) are presented as components, executing as location-independent entities within a distributed system. EXC3ITE currently supports the Common Object Request Broker Architecture (CORBA) [1] and is being extended to accommodate other middleware technologies such as the Distributed Component Object Model (DCOM) and Java™ 2 Enterprise Edition (J2EE).

The current EXC3ITE network backbone is a high-bandwidth Asynchronous Transfer Mode (ATM) network operating over commercial optical fibre and satellite links using military grade encryption, but EXC3ITE concepts are intended to apply more generally. Setting up distributed applications to run over an impoverished network can be anything but easy. The performance of distributed applications can become unacceptable because of network limitations such as longer propagation delay, lower bandwidth and poorer reliability of data transfer. In particular, satellite-connected nodes suffer appreciable (1-2 sec) round-trip latency in communicating with other nodes, due to the height of geostationary orbits. In addition, they will typically have less bandwidth available than those connected over terrestrial links. The question arises as to how increased latencies and reduced bandwidths affect the efficiency of middleware. For example, significant hand shaking between nodes could have a considerable impact of the performance of time critical applications such as real-time interactive simulation.

Most middleware technologies have primarily been designed for the commercial environment where clients and servers are collocated on the same local area network (LAN), rather than the conditions one would expect in military environments. With the growing number of applications that use middleware as their infrastructure, it is becoming common that users request distributed services across wide area networks. Higher demands are therefore placed on the performance of middleware operating beyond LAN environments. Middleware, by its nature, operates invisibly. This invisibility, along with the difficulty of isolating middleware-related network traffic, discourages close examination of middleware effect on the network. Although the performance achieved through middleware may not be as good as when using low-level approaches [2], this approach should still need to provide acceptable performance.

Further, the battlefield of the future will be characterised by higher demands due to mobility. Seamless access to information will need to be available anytime, anywhere. Deployable military units will be dependent on rapid and reliable communication to fulfil their missions, particularly in a hostile tactical environment. The anticipated dispersed mode of operation will mean that much of the communications will be via satellite with corresponding issues of high latency. Since latencies will not improve substantially in the future, it is not clear to what extent high latency will compromise performance. In addition, the tactical links, especially those to the highly mobile nodes, will suffer from a poor underlying error performance.

While the use of CORBA in unreliable wide area networks is often questionable, message queuing technology seems to be capable of providing reliable delivery of messages between processes despite network or object failures. In this paper, we consider the following question: does the message queuing paradigm have better performance than CORBA under diverse network constraints? We attempt to identify those aspects of CORBA and message queuing middleware, specifically in relation to communications, that would determine its ability to operate in the military tactical environment.

This paper reports a performance comparison of CORBA and message queuing for a typical client-server interaction under various network conditions. The performance is characterised in terms of response time and throughput at the application level. We then attempt to identify problems that arise from these conditions in interactions between client and server. In so doing, we aim to gain insights into performance effects due to overheads in the processing hosts and transport over the underlying network. Section 2 describes general characteristics of the global Internet. Section 3 considers the programming environment for distributed applications and the overheads incurred due to CORBA and message queuing. Section 4 describes the test environment used in our experiments, and the results are presented in Section 5. Section 6 discusses the fundamental differences between these two technologies. Finally, Section 7 draws conclusions.

## 2. Distributed Applications over the Internet

The global infrastructure of the Internet embraces a mix of communications and computing technologies, as is typical for distributed applications in military computing environments. The Internet of the future will be characterised by additional demands due to mobility. Seamless access to information will need to be available anytime, anywhere. Distributed applications often come with a spectrum of distribution requirements. Users in a fixed location will continue to have almost unlimited access to information by means of high capacity communications, such as satellite, optical fibre and radio relay. Some communication technologies will, however, be denied to truly mobile users.

The quality of the Internet is basically represented by two important network variables: packet loss and network delay. Packet loss measures the reliability of a connection. Measurements of the Internet [3] indicate that 0% to 8% of packets are being lost globally.

Packet loss can be attributed to congestion, bit errors, and deliberate discard. Routers discard incoming packets that cannot be transmitted or stored in the buffer. Packet discard due to corrupted data is more likely on wireless and satellite links because of poorer link quality. Some networking technologies such as ATM guarantee resource reservation for traffic of higher priority. If there is less bandwidth to transmit all incoming packets immediately, some of the packets of lower priority are either stored in a buffer or simply discarded.

The round-trip delay on the Internet is how long it takes for a data packet to travel from one point to another and back. Round-trip delays on the global Internet range from 80 ms to over 600 ms. A typical round-trip delay is 80 ms when going between national optical fibre backbones in North America. Higher latency connections are found on international and satellite communications. It typically takes 300 ms for data to reach its destination on the international Internet and for the response to be returned. It takes 250 ms over satellite links just for the signal to get from the earth to the satellite and back. A typical round-trip delay is 600 ms when a satellite is in the forward and return paths between client and server. As an indication, Table 1 reveals the round-trip delay of current Internet connections.

**Table 1: Typical Internet round-trip delay**

| | |
|---|---|
| local networks | 10 ms |
| national networks | 80 ms |
| international networks | 300 ms |
| satellite links | 600 ms |

As the Internet becomes the key to future distributed applications, the challenges to the designers are enormous. The network latency is substantial and the packet loss is high. These applications need to be robust and responsive even if the underlying network is slow and unreliable.

## 3. Programming Environment for Distributed Application Development

The principle architecture for encapsulation of a number of heterogeneous underlying infrastructures behind the same APIs is to divide the processing functionality into two different parts: the server-side processing and the client-side processing. The benefit with such a split of functionality is the performance gains that can be made. The drawback is the increased work implementing communication between the two parts. Communication can be implemented with custom marshalling or a wide range of middleware, such as RPC, sockets, CORBA and message queuing middleware.

Remote procedure calls (RPCs) are the most popular choice for designing distributed applications. The client component of an application simply invokes a function call to access the server component on a remote system, without having to specify the network address or any information. However, the RPC paradigm requires that the client and the server are always available and functioning, together with a reliable link between them. On

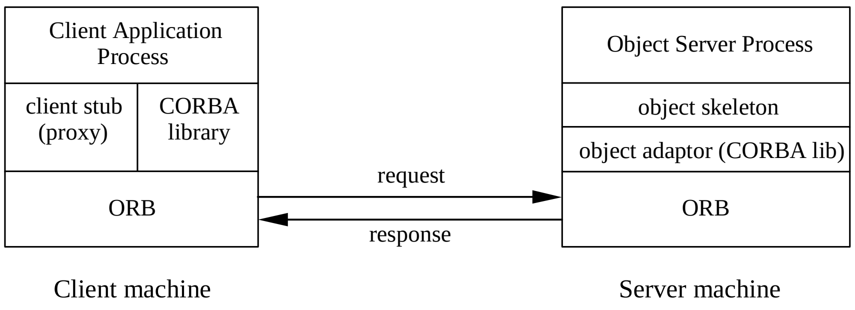

**Figure 1: CORBA Overall Architecture**

the other hand, message queuing allows applications to interchange information in a connectionless, asynchronous manner, which becomes a valuable building block in a distributed system.

### *3.1. Overheads due to CORBA*

As a descendant of the RPC mechanism, CORBA provides an environment for developing and deploying object-based distributed applications across a network. The overall architecture of CORBA is illustrated in Figure 1. CORBA supports location transparency to clients through stubs and skeletons. The exact connection mechanism between the client and the server is hidden from application programmers, as if the client and the server programs reside on the same machine. CORBA objects are accessed through public interfaces specified in an Interface Definition Language (IDL). An IDL compiler produces stub and skeleton classes, which are used respectively on the client and server hosts to provide presentation layer services for network transmission. Therefore the client and the server can be developed independently, which is a prerequisite for components to emerge.

When the client invokes a remote operation it is effectively invoking the operation of that name provided by the client stub. This involves marshalling and unmarshalling for transmission of the call parameters across different address spaces. Location transparency through stubs and skeletons adds considerably to the overhead. CORBA specifies the wire protocol for transmission between the client and the server running on different machines, with the ORB acting as an object bus. The ORB is a collection of libraries and network resources that is integrated with end-user applications. It is responsible for locating the remote object and preparing it to receive the request. The ORB then passes to the object the request in the form of a buffer so that the appropriate operation can be executed. The Internet Inter-ORB Protocol (IIOP) is defined for interoperability among TCP/IP based ORBs, in which the Internet is effectively used as an ORB communication bus.

To invoke operations on an object across heterogeneous ORBs, a client must first obtain its interoperable object reference (IOR). Once an IOR is obtained, the client-side ORB connects to the address and port number specified within the IOR. When the connection is successfully established between the ORBs, the client-side ORB creates a proxy object and finally returns the IOR to the client program. The client is able to invoke methods on the proxy object, which in turn, interacts with the server object. In effect, the client's method invocations are translated into operation requests that are sent to the server object.

When the client invokes the remote operation on the proxy object, the stub obtains a buffer and marshals the operation name and necessary parameters into the buffer. This buffer is handed to the underlying ORB, together with the IOR. The client-side ORB then sends the buffer to the target server through the established connection. After receiving the buffer, the server-side ORB unmarshals the IOR to determine the requested object. The remainder of the buffer is then passed to the object skeleton. The skeleton unmarshals the operation name, and the parameters for the operation. It then invokes the operation of the object, passing it these parameters. Once the operation has been executed, the skeleton obtains a buffer and marshals any results that are returned from the operation into the buffer. The buffer is then passed to the ORB to be sent to the client. The proxy object then unmarshals the results, checks for exceptions, and returns them to the client program.

CORBA processing overheads can be attributed to many factors [4]: presentation and session layer conversions and data copying, server demultiplexing, and buffering for network reads and writes. In the process of object activation, a client always binds with an agent or a directory service. This ensures that the request is always routed to an active server, allowing object migration and load balancing. On the other hand, if a client is given a persistent IOR, it needs to contact the implementation repository to ensure that the first request for an object is passed to the correct server. Once the client knows how to contact the object, further requests can bypass the access to the repository. Nevertheless, the location-forward reply from the repository inevitably incurs additional overhead in the retransmission of requests, especially when requests carry large amounts of data.

Depending on the network delays between client, agent, implementation repository and server; significant delay is entailed in the process of object activation. The IIOP consists of simple request-reply interactions. In addition to the TCP and IP headers, IIOP messages typically carry IIOP headers to support ORB interoperability and CORBA transparencies such as presentation layer translation of data into an interoperable format. Our observation of data transfer using Borland VisiBroker ORBs indicated that the IIOP overhead is relatively constant at 60 bytes from client to server and 28 bytes return, regardless of the size of IIOP messages. This seemingly insignificant overhead can, however, become a real burden to short IIOP messages for interactive applications especially over high-latency low-bandwidth channels.

The mainstream of CORBA-based solutions is targeted at LAN-based applications and relies on fast, reliable connections. Mobile and satellite links tend to display more variation in operational behaviour that CORBA may not adequately accommodate. Mobile end-user systems often experience changing characteristics of the underlying communications infrastructure — station closures, variation in throughput, and inadequate coverage — leading to a significant number of lost packets. Often distributed applications are not written to deal with performance degradation of the underlying networks. In this

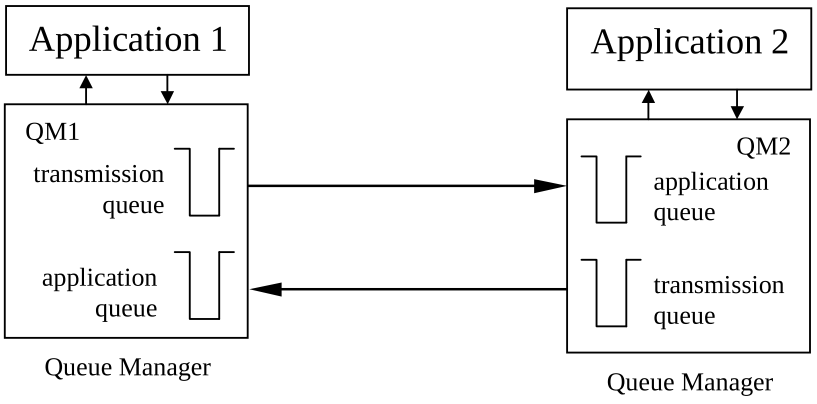

**Figure 2: Message Queuing Middleware**

environment, objects can fail unexpectedly, and the response time for a method invocation can be unacceptable.

### *3.2. Overheads due to Message Queuing*

Message queuing allows a number of application processes to exchange messages through an independent communication infrastructure, as illustrated in Figure 2. Message queues are created on all of the machines participating in the distributed application, so that each application can be built and executed independently. A message is a unit of information that is sent from a process running on one computer to other processes running on the same or different computers on the network. There is however no IDL or no marshalling in message queuing. A message is just a string of bits and it is up to application developers to ensure that the sender and the receiver understand the message layout. Transparent to the application, local queue manager holds messages in a message store. If the messages are destined for a remote queue manager, the local queue manager stores them on a transmission queue until they can be successfully transmitted and stored at the remote queue manager.

The queues are independent of program. Many programs can put messages into the same queue or get messages out of it. A program can also access multiple queues, which are controlled by a queue manager. The global functionality depends on how individual processes are assembled by configuring queues and communication channels into message flows. When a program attempts to send a message, the message is first moved into its queue. The middleware does the transfer of messages from the sender's queue to the receiver's queue on the best possible path. It also allows for delayed messages, both delayed delivery from a sender and delayed pickup from a receiver. A persistent queue keeps logged on disk so that if the system goes down, the queue is not lost. It ensures that, whatever happens to the network or the target computer, the message is eventually placed in the destination queue. Although a queue represents a first-in, first-out store, a receiving program can also remove messages out of order, and priorities can be used to implement various qualities of service. The best-known message queuing middleware includes MQSeries from IBM and MSMQ from Microsoft.

Due to its flexibility and persistence, message queuing allows a distributed system to be made of a large number of processes while reducing the complexity. It allows messages to be temporarily stored in queues before delivery, at the pace with which the receiving processes can read. Request and response messages have to go through a series of queues before they arrive at their destinations. This loose coupling implies that the behaviour of each application process is fairly independent of the state of other processes. An important feature is that interacting processes do not need to be aware of each other at run time. The robustness of the system is therefore greatly increased. They only need to know the standard interfaces and the queues they are going to use for message exchange. This style of communication is particularly suitable to mobile computing devices that are often disconnected from the corporate network. In contrast, processes using RPC-based middleware are always aware of the status of their counterparts and often have to take corrective action if a counterpart fails. This creates a number of failure modes and makes the system more brittle as it grows.

An application can be designed to be synchronous using message queuing middleware, resembling the client-server paradigm. Technically, it is an asynchronous application that expects a timely response to an originated message. The client sends a request message to an input queue. The server retrieves the request message from the input queue with unlimited wait. After processing by the server program, response is placed to a reply queue. The client attempts to get a response message from the reply queue with indefinite wait. In that case, the response time contains different components: processing time of the application at both ends, overheads of the message queuing middleware and operating system, waiting time for message retrieval, and transmission time of request and reply messages over the network.

Message size is a key component in message processing, primarily due to data movement within the queue manager. Large messages are subject to additional processing overhead. Data must be moved out of the application and into the buffers of message queuing. Also, it must be logged on disk if persistent, non-persistent messages are however not guaranteed delivery. Besides the message volume, the performance of message queuing middleware is also dependent on the characteristics of the underlying network, namely, bandwidth, latency, reliability and network traffic.

## 4. The Experimental Environment

The principal aim of our work was to compare the performance of distributed applications over CORBA and message queuing in a variety of network environments. Each network condition was designed to represent different degrees of connection performance. Figure 3 illustrates the test environment, which was set up such that the

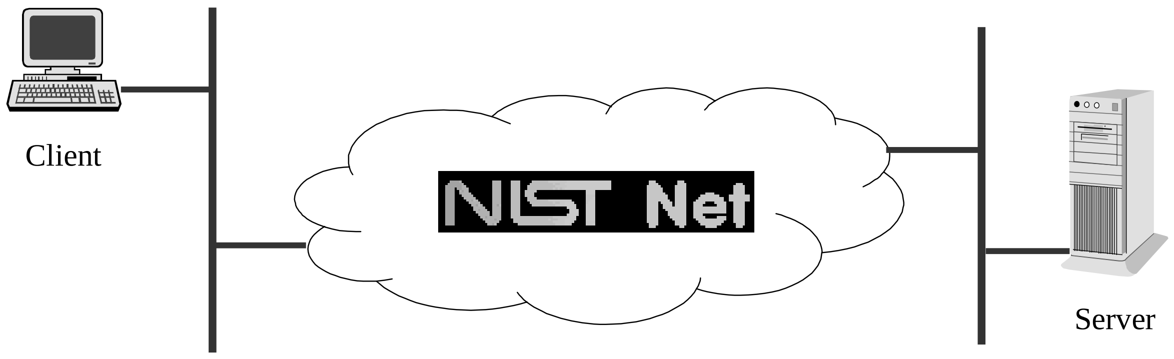

**Figure 3: The Experimental Environment**

experiments were conducted in identical environments except the middleware layer and the corresponding APIs in the test programs.

The client is a Pentium III 500 MHz machine running Microsoft Windows 2000 Professional. The server is a Pentium III 866 MHz machine running Microsoft Windows 2000 Server. All tests were performed on these two machines. Various network conditions were simulated using the NIST Net Emulation Package [5]. The client and the server were connected to the emulated WAN via T1 links of 1.544 Mbps, representing a typical business connection to the Internet. Borland VisiBroker for C++ [6] was selected to represent an implementation of CORBA, while IBM MQSeries for Windows 2000 [7] was used to provide the functionalities of message queuing middleware. TCP/IP was used as the wire protocol for transmission between the client and the server. TCP/IP for Windows 2000 has a number of features that optimise TCP performance in various network environments, including support for large TCP windows, selective acknowledgements, and better round-trip time estimation.

The performance measures of interest are the mean client response time and the throughput. A simple C++ test routine initiates remote invocation over middleware so that data flows in both directions alternately in a ping-pong fashion. We measured the response time and throughput between the client and the server for a range of message sizes, under varying network conditions. The response time is the actual time that elapses between the initiation of a method invocation by the client until the result is returned to the client. This simple operation was repeated 2000 times to get a reliable average value and the available throughput as a function of the message size.

The test program for MQSeries was written to send a message of a fixed size to the queue manager on the server, which then transmitted the message back to the queue manager on the client. The client and server machines were grouped into one cluster so that the two queue managers were logically associated in some way. To be comparable to the experiment for CORBA, the experiment for MQSeries was conducted in express delivery mode, in order to avoid disk access delays that would otherwise reduce performance. In express delivery mode, messages are only stored in memory until successful retrieval, and are not written to disk. They are therefore unrecoverable should the system crash before they are retrieved. The test program was designed to behave synchronously, such that its performance was gated by its round trip response time. It had to wait for the return of a transmitted message before sending the next one.

We used the same ping-pong test throughout the experiments. The request service time at a particular message size is basically the same for all the tests under different network conditions. By using different combinations of network latency, packet loss, and message size, it is possible to investigate the effects of these variables on the overall performance of the application.

Various network conditions were selected for the collection of experimental results. End-to-end network latencies of 5 ms, 40 ms, 150 ms and 300 ms were simulated to cover the range of round-trip delays expected in typical Internet connections. We simulated packet losses from 0% to 8%, which is the range expected in the Internet. The packet loss is the same in both directions the data blocks are being transferred. Performance results for different data block of size 1 to 65536 bytes were collected.

## 5. Results

Figure 4 - Figure 7 compare the response times between the client and the server across the underlying emulated network for various message sizes. Each graph shows the response time as a function of message size for a particular value of network latency.

Performance degradation is noticed when the client and the server have to communicate under worsening network conditions. It is obvious that the response time increases steadily with the value of network latency (from 5 ms to 300 ms) as well as the message size (from 1 byte to 65536 bytes). However, the effect of network latency on the overall performance is not as detrimental as that of packet loss. A small level of packet loss can greatly increase the response time. Note that high packet loss is not uncommon in long-haul connections where wireless or satellite links are involved. The difference between CORBA and MQSeries is not significant. However, MQSeries is more sensitive to network effects than CORBA. When network conditions are favourable at low latency (5 to 40 ms) and small packet loss (0 to 2%), MQSeries exhibits better performance than CORBA. The performance degradation of MQSeries is more noticeable than that of CORBA under poor network conditions, either at prolonged network latency (150 to 300 ms) or at high packet loss (4 to 8%).

In an ideal situation, the cost of sending a message between two programs located on different processors can be represented by two parameters: the message start-up time $t_s$, and the transfer time per data unit $t_w$ [8]. The former is the time required to initiate the communication, whereas the latter is determined by the effective bandwidth of the communication channel linking the source and destination processors. Therefore, the time required to transfer a message of size $L$ units between two processors is:

$$T_{msg} = t_s + t_w L$$

According to this cost equation, we can observe the general trends of data in Figure 4 to Figure 7, each of which is basically linear with some small fluctuations predominantly in the early part of the data series. This can be attributed to the effect of segmentation overheads at the IP layer when the message size $L$ is relatively small but larger than the path maximum transmission unit (MTU). The MTU is a link layer restriction on the maximum number of bytes of data in a single transmission. When a message is too large to be sent across a link as a single unit, a router may fragment the message. This effect is less significant as the message size increases. In our experiment, an MTU of Ethernet (1500 bytes) was used.

Another interesting observation is the spread of data series at various values of packet loss. Results from CORBA show that small packet loss (0% to 2%) does not greatly affect the response time in local networks at 5 ms latency. It becomes more significant when the network latency increases as in Figure 6 and Figure 7. The spread of MQSeries is generally more severe than that of CORBA especially at lower network latency, due to its vulnerability to packet loss. It should be noted that as the network quality deteriorates, the recovery algorithm at the TCP layer takes longer time to successfully transmit the whole segment across the network; as a result the TCP mechanism increasingly dominates the response time.

In the ideal situation, $1/t_w$ is the maximum effective bandwidth or throughput, which is only achievable as the message size approaches infinity. The throughput in different network settings of our experimental results is shown in Figure 8 - Figure 11 as a function of message size. In general, the throughput decreases as the network latency or packet loss deteriorates. We observe ripples of varying degree in the data series as the throughput increases. As mentioned, the application transfers data as a sequence of smaller blocks (in path MTU packet size), rather than attempting to transmit the entire block as a unit. These results demonstrate that varying the size of data block may have significant effect on the throughput. The throughput is generally poor for small message sizes, but improves for larger sizes. The performance of middleware is extremely sensitive to the message size, with small variations in the message size producing different throughput. This behaviour is due to the Nagle algorithm, as used by TCP to buffer and aggregate small packets. The Nagle algorithm is effective in preventing flooding of wide area networks with very small packets. However, when used on networks of lower latency, it can have a disastrous effect on the response time of certain applications. For such applications, the only solution is to disable the Nagle algorithm.

As a note of caution, these results were generated according to our experimental set-up (both hardware and software) and the design of our test programs. The actual values of these results are not meant to indicate the performance of other distributed applications over CORBA or MQSeries even in the same network environments.

## 6. Fundamental Differences between CORBA and Message Queuing

CORBA and message queuing are two major categories of middleware technologies. CORBA requires marshalling and unmarshalling at the sender and receiver respectively, while message queuing involves message transfer to and from queues as intermediate buffer. The processing overhead of CORBA is somewhat independent of the quality of the network. In contrast, the processing overhead of message queuing is a function of the number of message switches along the path. In addition, their fundamental differences are in the type of programming methodologies and communication primitives they make available.

### *6.1 Communications Mode*

The most basic attribute of a communication mechanism is whether the communicating applications exchange information in a synchronous or an asynchronous manner. In synchronous communications, producers and consumers execute in a coordinated fashion, with producer/consumer pairs cooperating in data transfer operations. In this manner, an application establishes a connection to its peer, sends a request to the application, and waits for information to be returned by the peer. An application cannot send information if its peer is not ready to receive. It is also blocked from performing other work until a response is returned. Hence both applications are simultaneously engaged in the communication process, and the respective machines become tightly coupled. The applications built on top of conventional RPC tend to assume that all the hosts in a network are always reachable and that the network is constantly available with sufficient bandwidth. CORBA, built on the RPC mechanism, facilitates sequential processing due to its synchronous nature between distributed objects. Program control is relinquished when data is transferred during each request. While the server performs the requested task, the client must wait for the response so as to regain the control. The client may experience unacceptable delay if the server is processing a large backlog of requests from other clients.

In contrast, message queuing is non-blocking, relying on asynchronous communication in which a consumer obtains data without the cooperation of the producer. Message queuing can offer higher levels of fault tolerance and availability because the queue acts as a buffering mechanism that allows either side to accomplish its work in the absence of the other. A client can continue to send request messages to a queue regardless of whether a server is running. Once the server becomes ready, it can resume processing request messages that have accumulated. The server can also continue its work after the client has stopped running. This type of asynchronous communications decouples producers of information from consumers. The corresponding machines are therefore loosely coupled.

RPC-based mechanism however requires a direct connection to be set up between a client and a server. In message queuing, program control is basically not transferred explicitly from one part of the application to another. This makes it easier to distribute control among an application's interacting components. Program control is returned to the client immediately after an asynchronous request is submitted. This allows distributed applications to be decoupled and executed in parallel, without having to wait for the response. The client application is able to process more parallel requests, though usually at the expense of increasing the response time. Likewise, the entire network path between the sender and the receiver may not need to be available when the data is in transit, supporting more flexible and time tolerant communications. In fact, messages are modelled as events in message queuing, instead of method calls as in CORBA. All applications communicate directly with each other using messages, which are only meaningful to the intended peers.

## *6.2 Location Transparency*

Modern middleware technologies are built on the principle of location transparency. The location of distributed components is hidden so that the system appears as an integrated computing facility. It facilitates application developers to design and implement at the function call level similar to the standard procedure invocation, regardless of the location of the object implementation. Like a normal procedure that provides the execution context by means of a stack, a remote procedure call builds a computing context and transfers the execution control to the called entity. It allows the remote procedure to execute almost as if it were in the same process as the caller. When a normal procedure returns, the control flow is transferred back to the caller. The RPC mechanism enforces this behaviour by suspending the calling thread after having sent the procedure call message and letting it wait for the return message. The caller, upon receiving the return message, resumes processing within the same context in which it sent the call.

CORBA hides the details of inter-process communication in its architecture. The code for an in-process object automatically works for an out-of-process object. The client-side code is not required to locate the object. Yet it requires both the client and the server to synchronise their interaction. It entails substantial complexity to ensure the server is available when a request is made, and the client is still running when the server responds. Synchronous interactions simplify the development of distributed applications by supporting an implicit request/response protocol that makes remote operation invocations transparent to the client. Everything works as if the call is a normal procedure call; the only problems are in the management of communication and process failures. This makes RPC-based middleware inappropriate in the event that servers or clients are only intermittently available. In that case, asynchronous messages can be delivered from suppliers to consumers without requiring the participants to know about each other explicitly.

In reality, location transparency of remote objects comes with some performance overheads. The client and the server are required to participate in numerous interchanges at the transport layer in order to carry out a simple interaction. Information interchanges across high-latency connections are expensive, causing dramatic performance degradation when LAN-based applications are deployed to wide area networks.

### *6.3 Programming Complexity*

Converting an in-process object to a distributed application is straightforward in a sequential programming environment. Using somewhat identical interfaces, distributed program components executed on different computers can logically communicate and synchronise in sequence. Since the entire computation moves sequentially from one computer to another, many sequential programming techniques can be used unchanged. The only principal concerns are that the routines must be able to deal with a variety of data distributions and that implementation details such as data structures and communication operations be hidden behind interfaces.

Parallel programming is intrinsically more difficult than conventional sequential programming. It is hard to manage explicitly the execution of multiple processors and coordinate the inter-processor interactions. A distributed application running asynchronously is required to specify the individual components that are to execute concurrently, producer/consumer relationships between components, and the mapping of components to computers. It allows computation to be overlapped with communication. In message queuing, all applications communicate directly with each other using messages, which are only meaningful to the intended peers. Actually, polling can be an expensive operation on some computers, in which case application developers must trade off the cost of frequent polling against the benefit of rapid response to remote requests.

In addition to traditional synchronous calls, most RPC-based middleware supports non-blocking calls. An example is CORBA messaging specification that establishes the requirement for asynchronous support by the ORB. The client application simply calls on an asynchronous call. The call is routed as appropriate to the target server and is queued, and the server resources are invoked as needed. The client may disconnect from the ORB, who may connect again at a later time and request to receive the results. This scenario is extremely useful in support of mobile clients. However, using RPC-based middleware asynchronous is significantly more complex than synchronous use. Asynchronous calls typically involve multitasking, polling, or callbacks — all of which add to the complexity of the application.

## 7. Concluding Remarks

We have attempted to compare the performance of CORBA and message queuing in a range of different network environments, in terms of response time and throughput at the application level. The performance of distributed applications running over middleware is sensitive to network latency and message size. It is even more sensitive to packet loss. Any unreliability in communication can make a system difficult to use — even a normally insignificant level of packet loss can severely degrade an affected application. As a consequence, reliable transmission is extremely important for delivering high performance middleware-based distributed applications to end-users.

Performance of distributed applications can be characterised by the quality perceived by the users. With the increasing use of middleware in mission-critical applications, performance issues become very important. In particular, response time is critical in interactive applications, with a target that is usually less than 2 sec. Beyond a typical threshold around 4-5 sec user complaints increase rapidly. For instance, a threshold for one-way delay for voice applications appears at about 150 msec. Beyond this level, the delay causes difficulty for people trying to have a conversation and frustration grows.

The quality levels for packet loss are found to be even more important. Some applications may be able to tolerate a certain level of packet loss. However, beyond 4-6% packet loss, video conferencing becomes irritating. The occurrence of a long delay of 4 sec or more at a packet loss of 4-5% is also irritating for interactive activities such as telnet and X windows. Packet losses beyond 10-12% are unacceptable: there are extremely long timeouts, connections start to be broken and most applications become unusable.

Middleware allows software objects to be executed in a location transparent way. Location transparency offers great flexibility for service creation, but as the software must be executed somewhere in the network on nodes of finite capacity, performance problems can arise due to inefficient placement of objects causing either overloaded nodes or excessive and unnecessary internodal communication. CORBA is a distributed object middleware based on a synchronous interaction style, requiring a high degree of synchronisation between the client and server. Our results indicate that the performance difference between MQSeries and CORBA is not significant. However, synchronous communication is too sensitive to variations in network quality to support distributed applications. Running CORBA-based distributed applications over tenuous networks can easily lead to unacceptable fragility.

Reducing communication overhead is crucial to harnessing the potential of distributed applications outside LAN environments. We need to pay careful attention to the granularity of objects and the number of logical transactions between client and server if acceptable performance is to be achieved. This is because each communication incurs not only a cost proportional to the amount of data transferred but also a fixed start-up cost. High start-up costs may suggest further agglomeration of objects so as to increase granularity. As our results indicate, the cost of distributing remote objects increases dramatically as the extent of connections grows beyond national bounds. Thus, it is necessary to perform dependence analysis for distributed applications and data-flow analysis on individual components to systematically optimise communication, and to improve placement of object implementations.

Current middleware technologies assume a high-bandwidth connection of the components, as well as their constant availability. In military networks, in contrast, unreachability and low bandwidth are the norm rather an exception. While CORBA is basically designed for tightly coupled objects using a synchronous style of communication, message queuing is more suitable for loosely coupled objects using asynchronous communications to provide reliable delivery of messages despite network or object failures. Since CORBA assumes that both client and server are available at the same time, message queuing should be used in case the client and server may run at different times.

The selection of different middleware technologies cannot considerably improve the response time for remote object invocations. Sadly, this is the indisputable fact that we have to tolerate. Nevertheless, we can significantly improve system throughput and availability by providing reliable asynchronous communication between computers that might become disconnected for some reasons, though at the expense of higher complexity in programming.

## 8. References

[1] CORBA: The Common Object Request Broker: Architecture and Specification, Revision 2.3: Object Modelling Group, 1998.

[2] D. Miron, and S. Taylor, "Performance Characteristics of a Java Object Request Broker," DSTO Report DSTO-TR-0696, June 1998.

[3] Internet Traffic Report, Andover News Network, http://www.internettrafficreport.com

[4] A.S. Gokhale, and D.C. Schmidt, "Evaluating CORBA Latency and Scalability over High-Speed ATM Networks," IEEE Transactions on Computers, April 1998.

[5] NIST Net Home Page, http://snad.ncsl.nist.gov/itg/nistnet/

[6] VisiBroker for C++ Programmer's Guide, Version 4.0, Borland Corporation, 2000.

[7] MQSeries Using C++, Fifth edition, IBM Corporation, 2000.

[8] Ian T. Foster, "Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering," Addison-Wesley, Feb 1995.

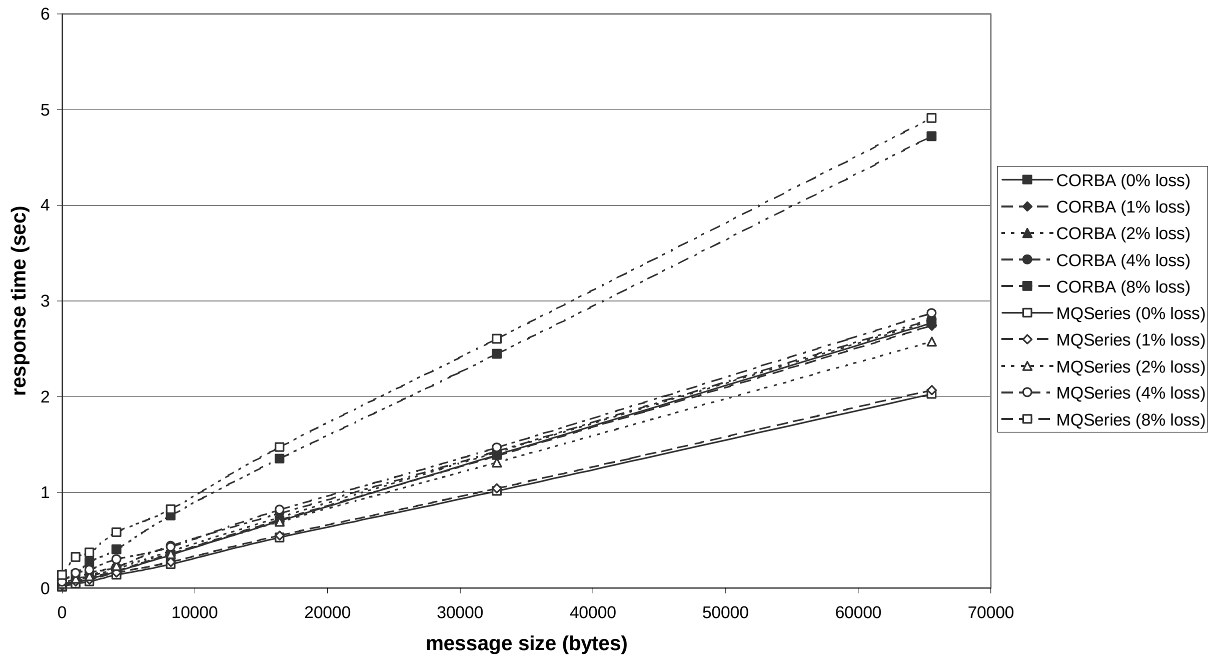

**Figure 4: Response time in typical local networks (network latency 5 ms)**

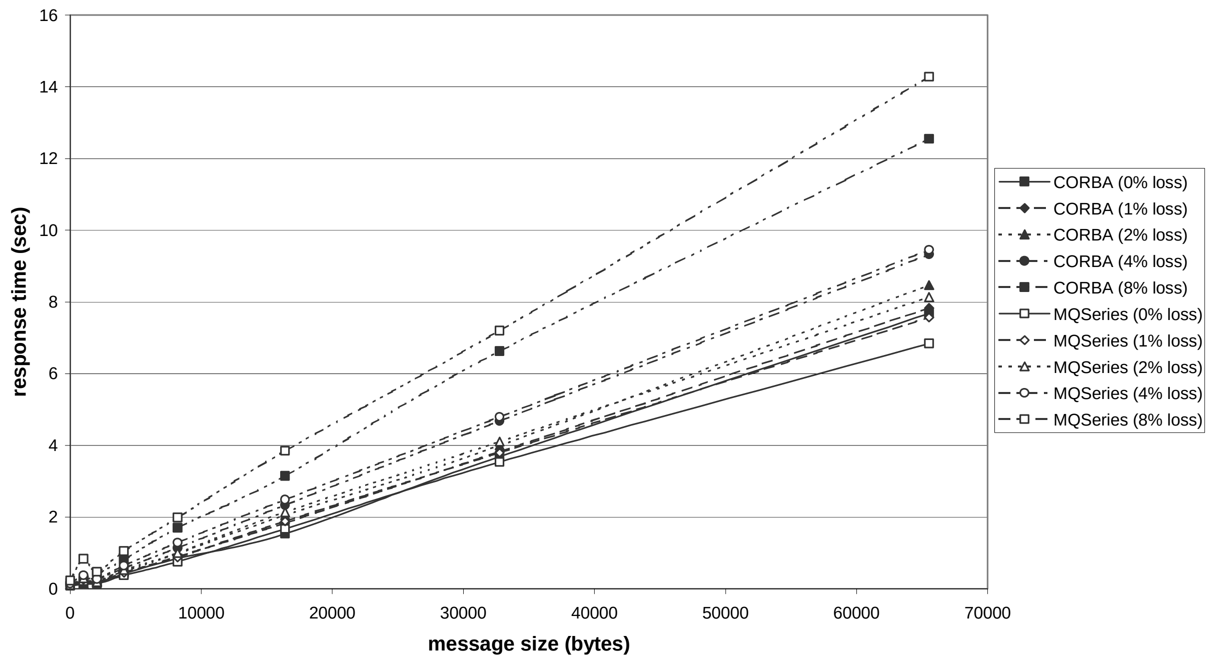

**Figure 5: Response time in typical national networks (network latency 40 ms)**

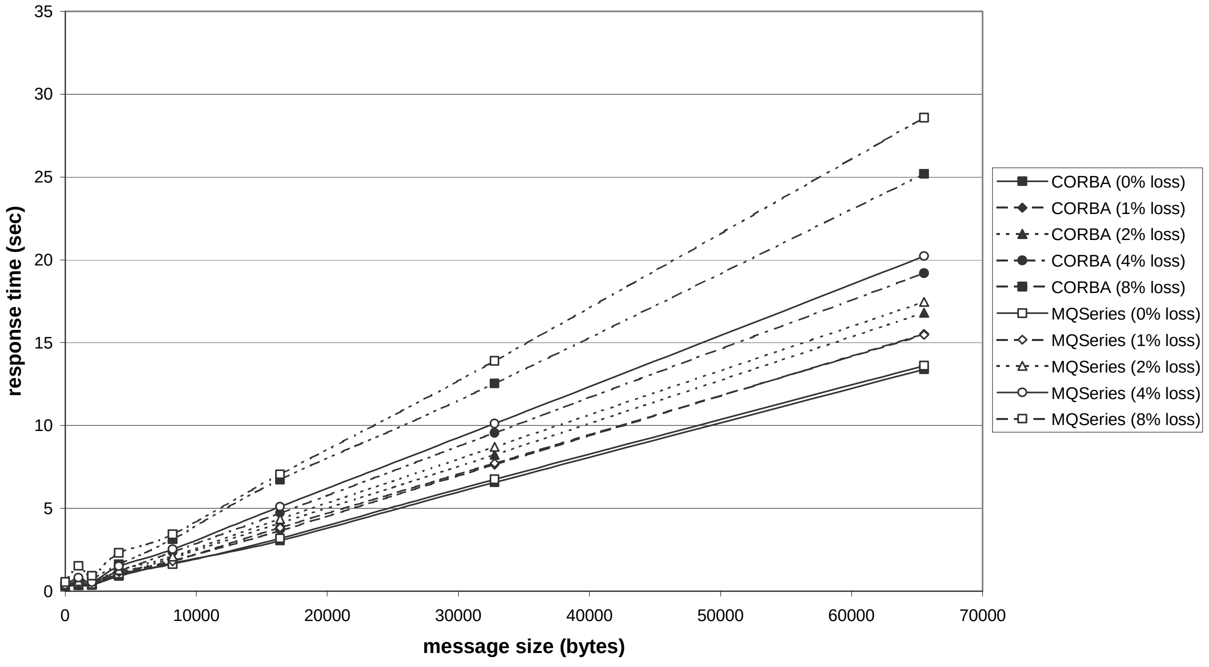

**Figure 6: Response time in typical international networks (network latency 150 ms)**

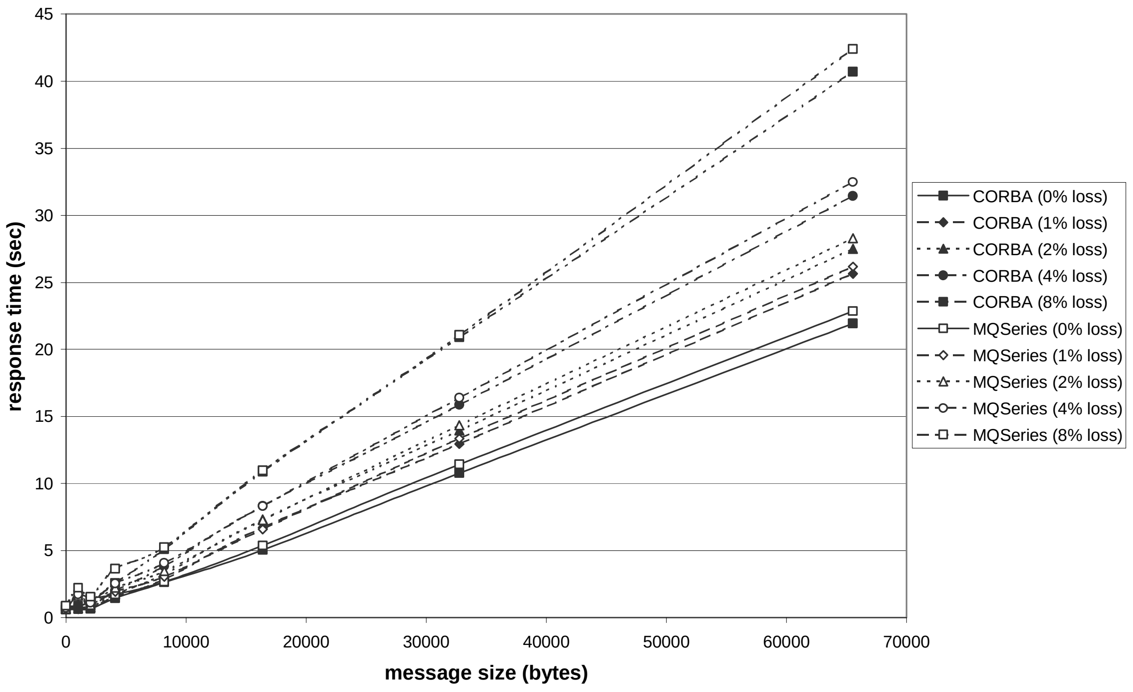

**Figure 7: Response time in typical satellite connections (network latency 300 ms)**

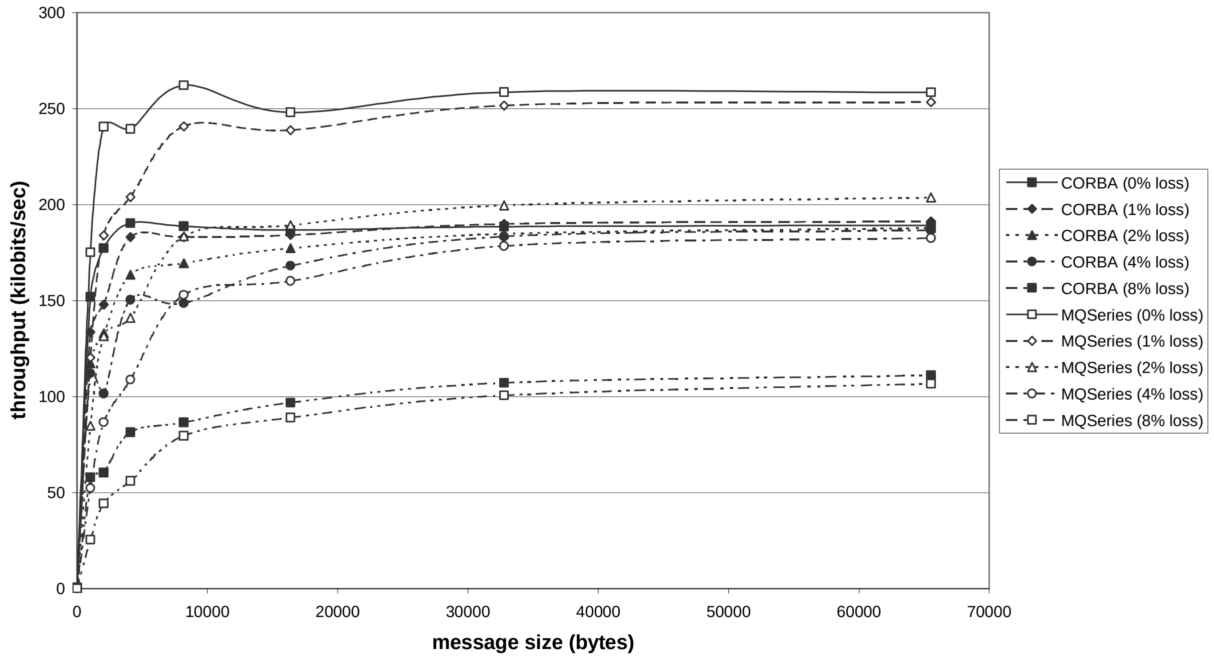

**Figure 8: Throughput in typical local networks (network latency 5 ms)**

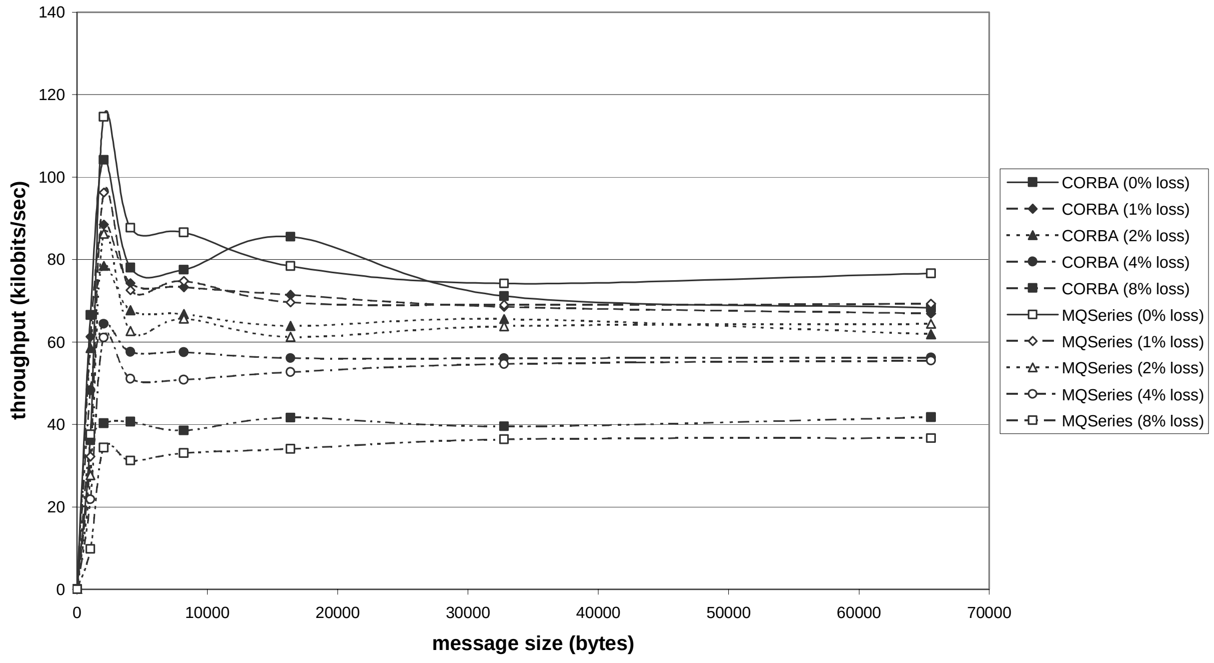

**Figure 9: Throughput in typical national networks (network latency 40 ms)**

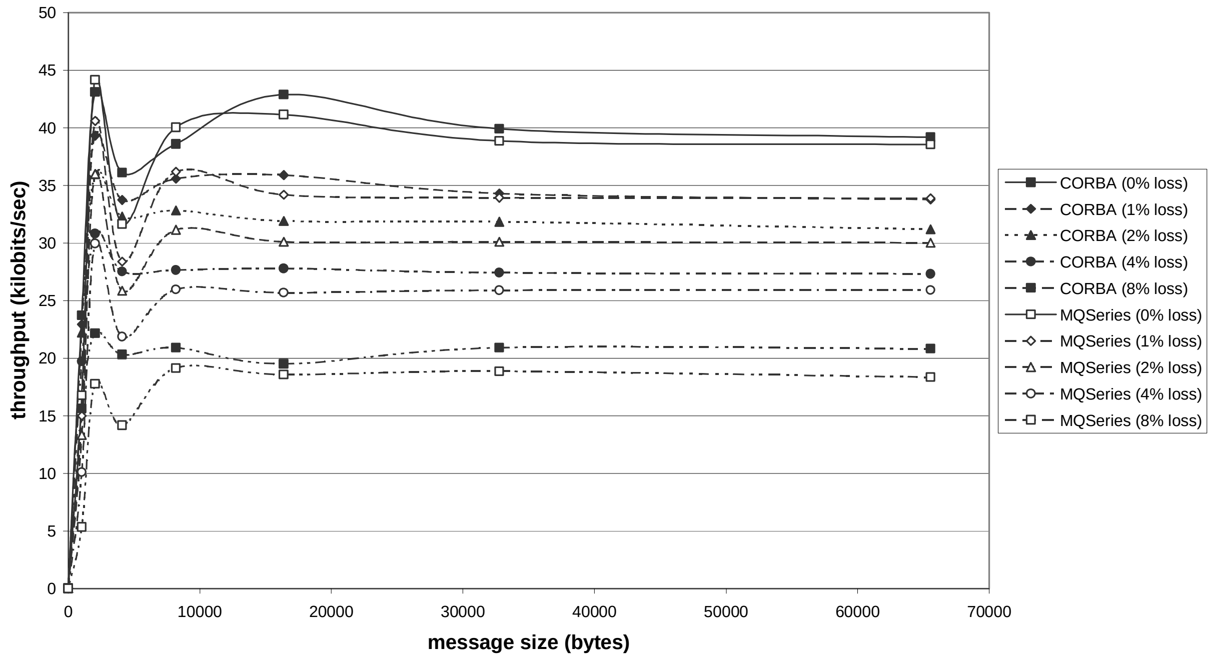

**Figure 10: Throughput in typical international networks (network latency 150 ms)**

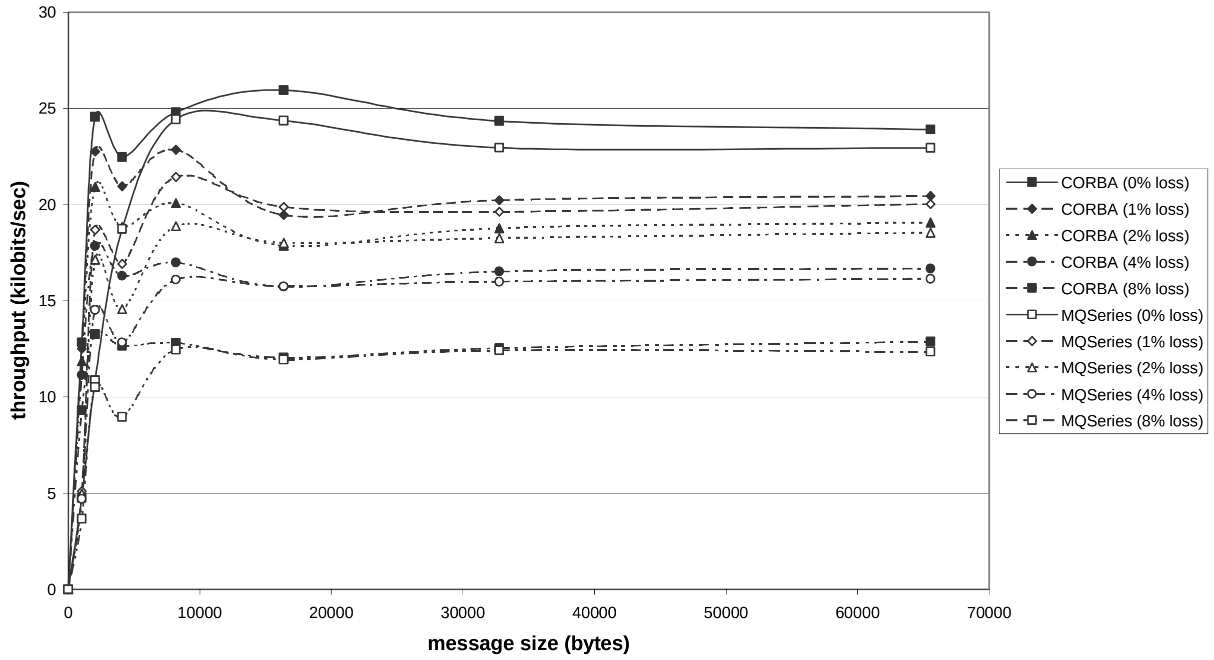

**Figure 11: Throughput in typical satellite connections (network latency 300 ms)**