Designing a CORBA-Based High Performance Open Programmable Signaling System for ATM Switching Platforms

Mun Choon Chan, *Member, IEEE,* and Aurel A. Lazar, *Fellow, IEEE*

Abstract—**ATM switching platforms are well suited for transporting multimedia streams with quality-of-service (QOS) requirements. This paper describes the system design of a high performance connection management system for xbind, a flexible open programmable signaling system for ATM switching platforms. The latency and throughput of call processing is improved by caching, message aggregation, and processing of requests in parallel. Using a set of general purpose UNIX work stations, we are able to attain a maximum throughput of close to 600 000 call operations/h (setup and delete) with an average call setup time of 85 ms. With a low traffic load of 3600 call operations/h, an average call setup latency of 11 ms can be obtained. The system is adaptive. By adjusting various control parameters, the connection manager(s) can be dynamically configured to trade off between throughput and call setup time.**

Index Terms—**ATM, common object request broker architecture (CORBA), open programmable signaling system.**

I. INTRODUCTION

DISTRIBUTED multimedia services, such as video conferencing, video-on-demand, and collaborative distributed environments, require flexible signaling platforms and transport networks that support quality-of-service (QOS). Due to the high bandwidth supported and connection-oriented nature, ATM switching platforms are well suited for supporting multimedia flows with long holding times.

On ATM switching platforms, information transport requires the establishment of a communication path between two endpoints. Such a task is performed by a connection management system that coordinates operations among a set of distributed software modules. While standards exist for the provisioning of ATM connectivity services [e.g., the user/network interface (UNI) and network/node interface (NNI)], many recent approaches (e.g., xbind [5], TINA [17], DCAN [6], and DCPA [7], [18]) provide network services by exploiting advances in distributed system technologies to give more flexibility to service creation and deployment. In these approaches, interactions among signaling entities are expressed in terms of high level operations. Signaling entities run on a general

M. C. Chan is with Bell Laboratories, Lucent Technologies, Holmdel, NJ 07733 USA (e-mail: munchoon@lucent.com).

A. A. Lazar is with the Department of Electrical Engineering, Columbia University, New York, NY 10027-6699 USA (e-mail: aurel@ctr.columbia.edu).

purpose distributed computing platform that provides an open and uniform access to abstractions modeling the local states of networking resources.

While these approaches meet the flexibility requirement, the use of general purpose platforms and programming languages [e.g., Java, C++, and common object request broker architecture (CORBA)] often leads to problems in performance. In [11], extensive performance measurements of various middleware implementations are shown. In order to improve the performance, two complementary approaches are possible. First, performance improvement can be achieved through optimization of the middleware implementation, independent of the application [3]. Second, application specific optimization can be performed. The approach in this paper falls in the second category.

The main contribution of this paper is to demonstrate how a high performance ATM connection management system can be built on a general purpose distributed platform, such as CORBA, and what kind of tradeoffs are involved. The three techniques used to improve the performance are: 1) application-level caching; 2) aggregation of connection requests; and 3) parallel connection setup. The first two techniques reduce the number of interactions among objects, and the last technique reduces the connection setup time. Asynchronous interactions are also used so that multiple connection requests can be processed at the same time.

The behavior of the throughput-delay characteristics of the designed connection management system is studied with respect to two control parameters. For various load conditions, these control parameters can be used to achieve the desired tradeoff among resource utilization, call setup latency, and call throughput. The system is implemented in C++/CORBA and operates on general purpose UNIX workstations. Its call throughput performance is comparable to the switching performance in the backbone networks for heavy traffic load ($\sim 10^6$ calls/h) and can achieve low end-to-end application-level call setup latency (~ 10 ms) for light traffic load. The results demonstrate that an open distributed platform can provide greater flexibility for service creation and allows the design of high performance connection management systems.

The paper is organized as follows. Section II describes the connection management framework, and Section III presents the measurement setup and a set of reference measurements. Section IV presents the performance results, including the behavior of the throughput-delay characteristics of the con-
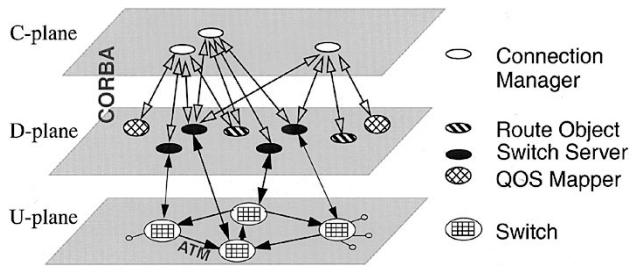
Fig. 1.   Object invocation in the xbind connection management system.

nection management system with respect to two control parameters.

II. DESCRIPTION OF THE CONNECTION MANAGEMENT FRAMEWORK

A. Description of xbind

Our approach to flexible service creation in an ATM switching platform is based on the concept of a broadband kernel. The broadband kernel incorporates an organized collection of interfaces, called the binding interface base (BIB), and a set of algorithms that runs on top of these interfaces. The principal aim of the broadband kernel is to provide an open programming environment that facilitates the easy creation of network services and mechanisms for efficient resource allocation. xbind is a prototype implementation of the broadband kernel concept [2]. In this paper, we restrict the discussion to the xbind connection management system.

Fig. 1 shows how objects are invoked in a xbind connection management system, where objects are organized in a planar structure. There is a clear distinction between the control (or signaling) plane and the transport (or data) plane. In the signaling plane CORBA objects communicate over an IP network. Due to their scalability and resilience to partial failure, IP-based networks are ideal for the transport of short control messages with little or no call holding times. On the other hand, the transport of multimedia with long holding times is ideally supported by an ATM switching platform. Due to the inherent stateful nature, an ATM switching platform can provide a much higher degree of predictability and is highly suited for the transport of streams with QOS requirements.

The lowest of the three planes is the data transport plane (the U-plane), which contains a set of interconnected ATM switches. The middle plane, the D-plane, exports a set of BIB interfaces that allows network resource to be controlled and monitored. Pertaining to the connection control are three object types, namely the QOSMapper, RouteObject, and SwitchServer. The QOSMapper provides the mapping of QOS requirements between the user and network domains. The RouteObject provides a path in the network between two endpoints, and the SwitchServer provides generic hardware independent interfaces to manipulate the resources on an ATM switch. These BIB interfaces are open. They are implemented on a CORBA/IDL platform and are the same regardless of operating system and hardware. For example, in our system, the same SwitchServer interface is exported

by ATM switches from four different vendors (FORE, NEC, ATML, and Scorpio). In general, multiple instantiation of these objects can be supported. A name service is used to locate the required service objects.

Connection management algorithms reside on the C-plane. Many classes of connection management algorithms can run on the set of defined BIB interfaces. The description of the connection management system described in this paper is only an example of a system that is tuned for high performance. For examples of other connection management systems implemented in the same framework, refer to [16].

B. Description of a Connection Setup

A graphic representation of a generic connection setup procedure is shown in Fig. 2. In step 1, a client application program, e.g. a video conference manager, sends a request to the ConnectionManager to setup a connection between the two hosts. The ConnectionManager maps the user-level QOS to network-level QOS. QOS abstractions for network resources may be defined for each traffic class using a specific cell loss and a cell delay requirement. Service abstractions for customer premises equipment such as PC's and workstations are specified in terms of frame rate and frame loss. In our framework, QOSmapper [4] translates the QOS specification specified in frame to QOS specifications specified in ATM cells and vice versa. This is shown as step 2. The resource specification is based on abstractions that are independent of the details of the system hardware.

In step 3, based on the QOS requirements, a path in the network is obtained from the RouteObject to connect these two endpoints. Routes are updated asynchronously by a Router object not shown in the figure.

Using the route obtained in step 3, resource reservation requests are sent in steps 4–6 to each of the SwitchServer objects. In the figure, only one ATM switch is shown (SS2). SS1 and SS3 are hosts. Resources are broadly divided into two groups: system resources (buffer, bandwidth, CPU cycles, etc.), and identifiers in the switch fabric for cell transport. The first group of resources are "homogenous." For example, 10 Mbit/s bandwidth on one switch is always "compatible" with 10 Mbit/s on another. As a result, reservation can be performed in a single step. This is, however, not true for switching identifiers.

Each ATM cell contains a header with routing information and payload. The two key information fields for switching purposes are the virtual channel identifier (VCI) and the virtual path identifier (VPI). In order to ensure the correct delivery of ATM cells, given the input port at which an ATM cell arrives and the input VCI/VPI field in its header, the cell must be routed to the correct output port, and the VCI/VPI header of the outgoing cell must have the appropriate value substituted. The process of setting up this information in the cell routing table is done in two phases and is illustrated in Fig 3. In phase one (steps 4–6 of Fig. 2), an output VPI/VCI pair is obtained from the output port of each of the first two ATM switches located in the path of the call. In phase two (steps 7–9 of Fig. 2), the output VPI/VCI pair of the upstream switch is mapped into the

CL - client program   CM - ConnectionManager   SS - SwitchServer
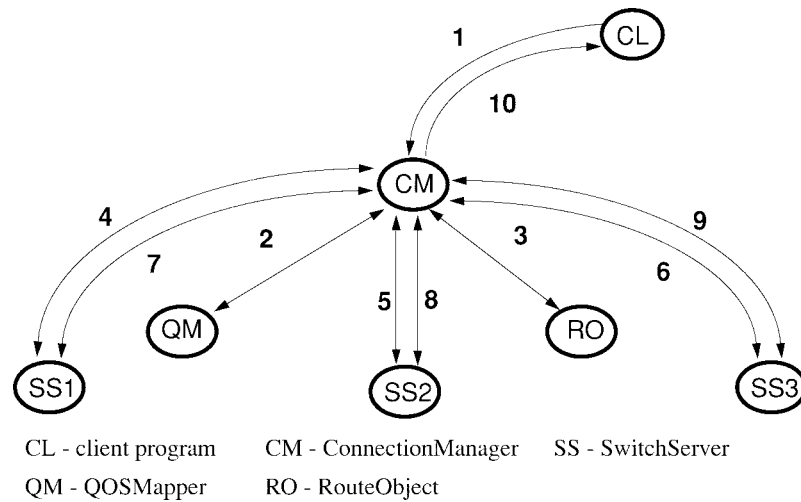QM - QOSMapper   RO - RouteObject

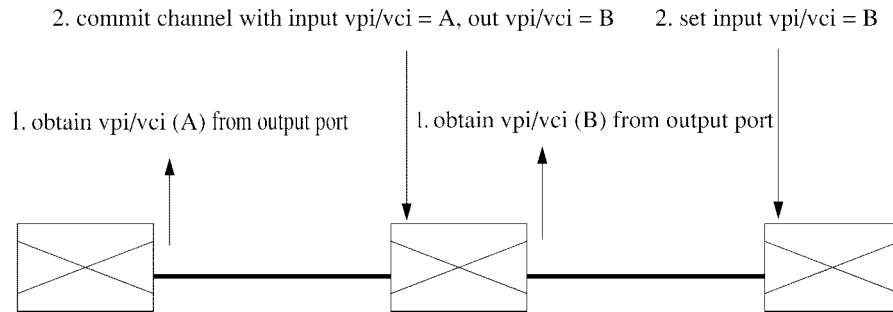Fig. 2. Execution of a connection setup request.

Fig. 3. Setting up the switching tables in an ATM switching platform.

input VPI/VCI pair of the downstream switch, and thereby, the channel is committed. In our description, the output VPI/VCI of the appropriate port of each intermediate switch is reserved first and then followed by the input VPI/VCI, but note that the reverse (input followed by output) is also possible. In step 10 of Fig. 2, the ConnectionManager returns the status of the connection setup to the client program.

C. Designing an xbind ConnectionManager

In a distributed environment, a connection manager has to take into account that the vast majority of the computations in a call setup are performed in the communications layer for processing of remote requests with small arguments. The processing can be inefficient in the following ways. Since the messages are small, the overhead of processing them can be large relative to the messages themselves. In addition, a significant portion of the total call processing time can be spent on waiting for remote operations to complete.

Based on these observations, we designed an xbind connection manager with the following features:

- caching of network states—store or prefetch resources so as to minimize the number of remote procedure calls;
- aggregate access to node server objects—aggregate access requests to remote objects as much as possible so as to reduce processing overhead;
- overlap communication and computation through parallelization—design the system to run with maximum

amount of parallelization so that processors can be kept busy as much as possible.

Our main objectives are: 1) reduction of the number of remote invocations and 2) the hiding of latency of remote invocations to the extent practically possible.

1) Caching of Network States: As shown in Fig. 2, the connection manager performs remote invocations on three classes of objects: the QOSMapper, the RouteObject, and the SwitchServer object. The performance of the connection manager is improved by caching for each of these objects a subset of their states. The five types of network states stored or cached are output (or input) switching identifiers, QOS mapping, route, bandwidth and buffer resources, and existing connection states.

For all these states, except for switching identifiers, the caching idea is pretty straightforward. Caching of states works well if the states cached have not changed, or in the case that they have changed, the out-of-date state might introduce some inefficiency but computation will still be correct and the state will eventually be updated (the so called soft-state). Caching of switching identifiers will be described in detail here, and the rest will be discussed briefly at the end of this section.

The problem of reserving switching identifiers is similar to a two-phase commit problem. All connected SwitchServer have to agree on the virtual path identifier/virtual connection identifier (VPI/VCI) pair to use. The idea for caching is to perform phase 1 of the reservation (steps 4–6 of Fig. 2) in

advance, instead of performing the computation on-demand. The VCI/VPI pairs obtained in advance are stored in a cache using the output port identifier and the switch identifier as the key. The result is that the connection manager obtains control over a set of available output VPI/VCI pairs that it requests in advance from the SwitchServers which can be used for any connection request whose path includes the specific output port and switch pair. During connection setup time, the connection manager simply looks for an available VPI/VCI pair with the correct key in its cache. If an available output VPI/VCI pair is found (a cache hit) for each switch/port on the path of the call, then the channel reservation process can be performed in a single step to the switch (phase 2 only). If no free VPI/VCI pair is available (cache miss), the normal two-phase operation is performed.

An alternative to performing the phase 1 only reservation is to perform both phases in advance, which effectively creates a number of end-to-end VC. The latter approach is similar to the VP approach and has the advantage of reducing signaling load in the network. No additional operation in the network is needed during connection setup if there are enough resources reserved in advance. The disadvantage is that resource utilization is not as efficient as a "pure-VC" approach, and there is a tradeoff between signaling load and resource utilization [12]–[14]. Our approach can be seen as an intermediate step between the "pure-VC" and the VP approach. Reservations are performed only locally, not end-to-end. How the resource is being used in an end-to-end manner is decided during connection setup time.

An additional comment is that if there are multiple copies of connection managers running, in order to avoid contention, all connection managers must agree on which portion of the name space to cache in advance. Therefore, cached either only the input port VPI/VCI pairs or the output port VPI/VCI pairs, but not both at the same time. Note that there is no restriction or coordination needed with respect to exactly what VPI/VCI ranges can be used. Locating the appropriate connection manager, QOSMapper, or RouteObject is a configuration issue and will not be discussed in this paper. Generally, such mappings are managed by object servers or traders.

By keeping the available VPI/VCI pairs on the connection managers instead of putting them in the SwitchServer that directly controls the switch, the resource state of the ATM network is partitioned and distributed to the higher level controllers (in this case, the connection managers). This partitioning and distribution process can be performed in two ways.

In the first approach, the number of VPI/VCI pairs reserved per port per switch is competitively decided among the controllers reserving the resources. Thus, each connection manager adjusts the number of entries (VCI/VPI pairs) kept in its cache depending on the call arrival and departure statistics and how much it is willing to pay for a low latency call setup. In the second approach, the size of the partitions is controlled by a distributed algorithm that attempts to optimize the partitioning of the name spaces on the network level. The two approaches differ in that, in the first case, the allocation process is performed using the rules of a competitive game,

whereas in the second approach, the partitioning process is performed in a cooperative manner. Combinations of the two approaches above are also possible. In Section IV-A, we study a competitive algorithm where a connection manager attempts to minimize the number of VCI's kept in the cache while trying to maximize cache hits.

For caching of routes, it is observed that during repeated call setups, patterns of call requests emerge that have the same source–destination (SD) pair. These patterns might include alternate routes between the same SD pairs. For calls belonging to these patterns, it is not necessary to have the connection manager contact the RouteObject each time a setup request is received. Instead, when the connection manager receives a route for a specific SD pair from the RouteObject, it will cache the route (or alternate routes) with a time stamp. Statistics of route selection for alternate routes may also be included. By defining a variable time out period for route invalidation of T s, let us say, the connection manager can reuse the "cached" route if it is less than T s old. If not, a new route will be requested. If the expected call throughput is 100 calls/s, and the probability of an SD pair appearing in a call request is 0.01, then by setting $T = 10$ s, there will be one update per 10 s per SD pair, instead of $100*0.01*10 = 10$ updates in 10 s per SD pair, an improvement by a factor of ten. Updates are performed on demand. Therefore, if a route "times out," there is no extra update.

QOS mapping information or policy, in practice, is relatively static and is unlikely to change at all during the lifetime of the connection manager. If this is the case, the QOSMapper can be created in the same address space as the connection manager, and accessing the QOSMapper from the ConnectionManager becomes a local invocation rather than a remote invocation. In the implementation, the "collocated" feature in Orbix is used to place the QOSMapper in the same address space as the ConnectionManager. In this way, there is almost no change to the rest of the code.

Caching of bandwidth and buffer resource is similar in nature to VP resource allocation, and caching of existing connection states for faster connection modifications and teardown is straightforward.

2) Aggregate Access to the SwitchServer Object: In a distributed object environment where the vast majority of object interactions are in the request-reply form with small arguments (<1 kB), the overhead incurred in processing these small messages can be substantial relative to the message size. In order to increase the throughput of the system, multiple requests are combined into a single remote invocation, as opposed to making multiple individual invocations, one per request. As a result, in a single invocation, the argument is a list of commands, each command corresponding to a single request. The number of requests (sum of add and delete) stored in the message buffer before they are sent out is a control parameter. We will refer to this parameter as the message threshold (T) parameter. This approach has the advantage that, by delaying the delivery of requests, the total number of remote invocations performed by the connection manager decreases for a fixed load. This is especially useful for invocations to the SwitchServers. Obviously, by delaying the delivery of

messages, the expected call setup time increases for low loads. For higher loads, the gain in reducing the number of remote invocations can outweigh the delay introduced by waiting. The message threshold parameter is used as a control for changing the system behavior for low load, where low latency can be obtained, or for high load, where latency is traded off for throughput. This tradeoff is studied in greater detail in Section IV-B.

In order for the aggregation scheme to work as expected, interaction with the transport layer has to be taken into account. In our implementation, the TCP_NODELAY socket option is set so that the Nagle algorithm [8] is disabled. The Nagle algorithm is a congestion control scheme (enabled by default for telnet or login sessions), which prevents TCP from sending less than a full-size segment when an ACK is expected for the connection.

3) Hiding Latency Through Parallelization: Due to the inherent delay incurred in accessing remote objects, a substantial number of processor time could be spent waiting or idling. In order to increase processor utilization, inter- and intrarequest parallelizing is used. Interrequest parallelization allows multiple connection request to be processed at the same time, and intrarequest parallelization allows a single connection to communicate with multiple switches at the same time. Both parallelization can be achieved by employing asynchronous object invocations, implemented as oneway calls in CORBA. Since oneway calls in CORBA do not guarantee delivery, a recovery protocol is needed in the connection manager to handle message loss.

Parallelization in various forms has also been studied in the context of ATM connection control in [16] and [18].

4) Functional Design of the ConnectionManager: All the design choices described in Sections II-C1–3 are integrated into the design shown in Fig. 4. The connection manager state machine is unaffected by the caching and aggregation schemes. In most cases, except for one remote access to the Switch-Server, most of the other calls are transformed from remote to local calls. For remote invocations to the SwitchServer, the requests are not delivered immediately. Instead, the messages to be sent are put in a message aggregation module, with one module per remote object. Therefore, aggregation is performed per target object. Messages in these modules are sent when either the number of messages reaches a particular threshold, which can be dynamically changed, or when a time out occurs. The message threshold parameter takes into account both add and delete requests. Time outs are used as a safety mechanism to ensure that messages do not remain in the queue for too long without being processed. They are needed when the traffic load is low to provide a bound for call processing latency. These time outs can also be set dynamically.

III. MEASUREMENT SETUP

A. Experimental Configuration

Fig. 5 shows the experimental setup for all the measurements. The network consists of two workstations, both HP9000/700 series with 120 MB of RAM, connected to an
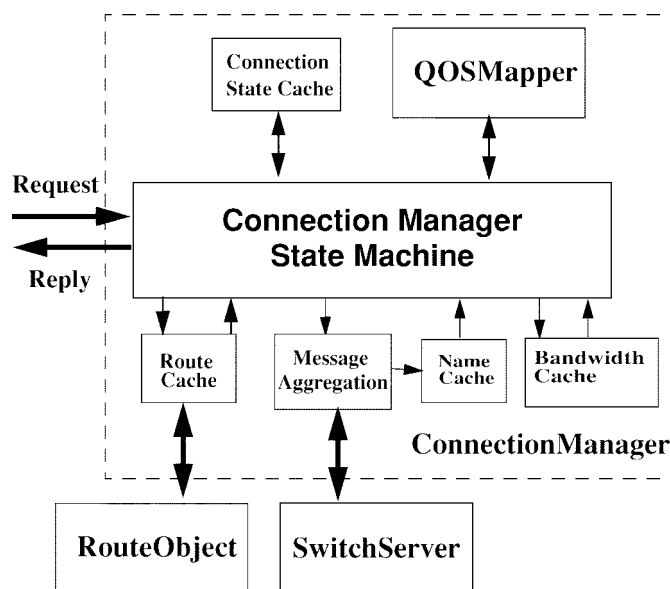

Fig. 4. Design of the ConnectionManager.

ATML Virata switch. A SwitchServer object runs on each of them. A third HP workstation serves as the controller for the ATML switch; the SwitchServer controlling the switch runs on top of it. This workstation communicates with the switch using the qGSMP protocol [11], a modified version of the GSMP protocol [9] with QOS extensions. All workstations communicate through a 10 Mbit/s ethernet LAN. The code is written in C++, and the CORBA implementation used is Orbix (version 1.3), from Iona Technologies. The ConnectionManager, RouteObject, and QOSmapper reside on the same machine, an Ultra2. The client application resides on a SUN Sparc20.

The experimental setup tests the connection manager performance in an ATM LAN environment only. Also, there is a complete separation of the control and transport network. The former is a 10 Mbit/s ethernet, and the latter is an ATM switching platform. The reason that signaling does not run on the ATM network is a matter of convenience and equipment limitations. Use of dedicated ATM connection for signaling could improve the performance results of our measurements but the improvement will not be significant because the bandwidth available on the LAN is sufficient for our purpose. Finally, the measurement setup is based on an actual video conference application. All software objects used for the connection setup experiment are the same, except for the video conference manager, which is substituted with a call generator (CG). Note that objects are distributed over five machines. The placement of objects to host are meant to demonstrate the effect of distribution. Any other placement is possible.

The order of executions during call setup requests is as described in the previous section. The client initiates: 1) a request to the connection manager; 2) followed by QOS mapping; 3) route selection; and finally, 4)–6) resource reservation. At the end of the execution, an end-to-end ATM connection is set up from the source workstation to the destination workstation through the ATM switch. In all measurements, the latency of
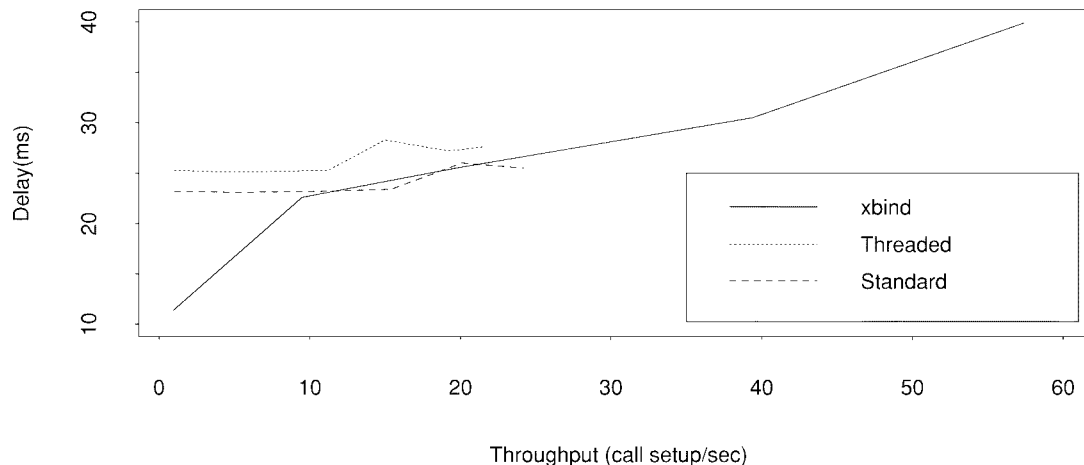
CLG - call generator   CM - ConnectionManager   SS - SwitchServer

QM - QOSMapper   RO - RouteObject   qGSMPd - qGSMP daemon

Fig. 5.   Execution of a connection setup request.

the call establishment is taken to be the period it takes to complete a call of `addConnection`() by the call generator (or client program) to the ConnectionManager.

Note that the performance measurement of interest is a complete call operation. Although the actual measurement results depend on the specific experimental environment, including the time it takes to execute a single RPC call, the underlying tradeoffs remain the same. An extensive measurement on the cost of remote invocation can be found in [11], and a study on the cause of performance bottleneck can be found in [3].

B. Baseline Measurement Results

The performance of a sequential implementation of the ConnectionManager is used as a reference point to be compared with later. This ConnectionManager performs steps 1–6 of Fig. 5, sequentially. All CORBA calls in this implementation are synchronous, which are easier to implement than the asynchronous implementation to be compared with later. All measurements are performed by repeating the specific call setup operations at least 1000 times.

The first measurement is call setup time, which is measured by sending only one add or delete request to the connection manager at any one time. The minimum call setup latency measured is 20.0, the average is 23.4, and the maximum is 63.3 ms. The 5 and 95% quantile are 20.5 and 31.0 ms, respectively.

The performance of a multithreaded version of the ConnectionManager is also measured. In this version, a separate thread is used to process each call request using a "thread pool" model. Measurements show that the multithreaded version, evaluated under the same experimental setup, is slightly less efficient than the sequential implementation. This is due to the overhead incurred in the locking of shared states and the context switching between different threads. The minimum latency for is 22.5, average latency is 24.9, and the maximum latency is 213 ms. The 5 and 95% quantile are 23.1 and 35.1 ms, respectively.

Early published connection establishment latency for point-to-point call establishment for one hop using UNI signaling

[1] has the following performance: minimum latency is 48.99, average latency 53.20, and maximum latency is 67.60 ms. Later improvements bring the latency for call setup latency for a similar LAN environment down to the range between 10–50 ms, depending on the switching software and hardware platform [10]. Since signaling runs on ATM VC's network and the call generator accesses the ATM adaptation layer directly, the execution overhead is expected to be lower then for a signaling system operating on top of CORBA. The result shows that a straightforward implementation of our connection setup model on an Orbix platform provides comparable performance.

The throughput-delay characteristics are also investigated. The call generator uses asynchronous calls, generating calls to the connection manager using a Poisson model. Successful call setups are released with a very small holding time of 0.5 s. A small holding time is used so that the total number of active VC's remains small even for a very high traffic load. [There is a need to keep the number of active connections small (<500) due to a limitation in the switch control software used. The usable VCI range is between 0–1023, and only VPI zero can be used.] Successful call setups are immediately torn down. The throughput is measured in terms of call setup per second. However, since both call setup and release are performed, the number of call operations (add and delete) is twice the throughput measured.

The results are shown in Fig. 6. Results show that when using only synchronous calls within the connection management system, the performance of both the sequential and threaded ConnectionManager saturates at a call arrival rate of about 25 calls/s.

IV. PERFORMANCE MEASUREMENT OF A XBIND CONNECTION MANAGER

In this section, we present the performance results of an xbind connection manager. The experimental environment used is the same as that described in Section III-B. Each point is the average of at least 10 000 calls.

Fig. 6. Throughput-delay curve of all three ConnectionManagers.
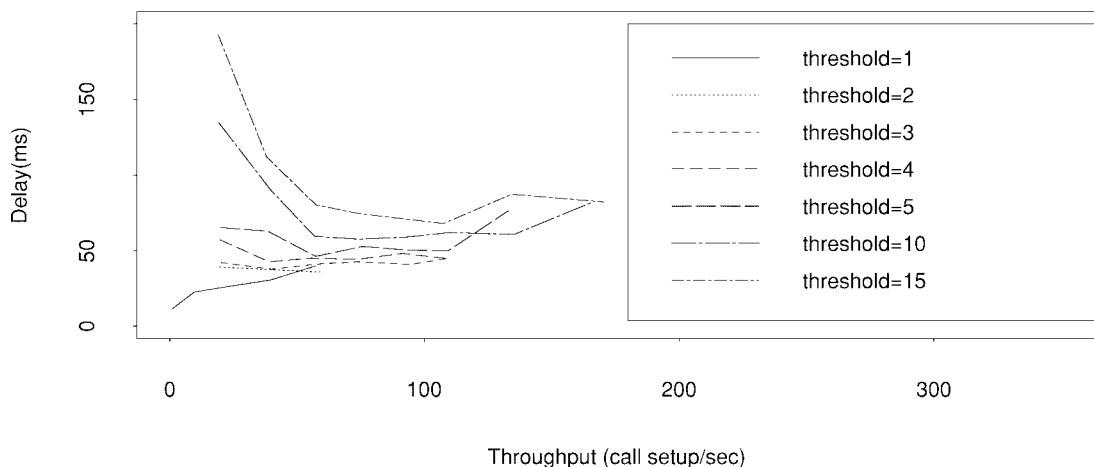

Fig. 7. Impact of threshold parameter (T) on throughput-delay characteristics of the xbind connection manager.

Fig. 6 compares the throughput-delay characteristics of all connection managers described so far, namely: sequential implementation, threaded implementation, and the xbind implementation. In the xbind implementation of the ConnectionManager, routes cached time out after 5 s, the message threshold is set to one (messages are sent immediately with no aggregation), and the number of VCI cached is made so large (>200) that all arriving calls execute steps 4–6 of Fig. 2 locally. Finally, messages that remain in the queue for more than 1 s will be flushed. By setting the message threshold to one, the performance gain observed is due only to caching and parallelism, not message aggregation. We will study the impact of message aggregation in Section IV-B.

The performance curve of the xbind implementation shown in Fig. 6 has three regions of operation. In the low load region (<10 calls/s), the latency is better than in the sequential implementation. The effect of caching and parallel connection setup for reduction of call setup latency is demonstrated here. In the second region with load between 10–25 calls/s, the performance of the xbind ConnectionManager is similar to that of the sequential implementation. This is due

to the fact that a single synchronous call is slightly more efficient than two asynchronous calls. Therefore, as the load increases, the inefficiency of asynchronous calls begin to offset the utility of caching and parallelism. Finally, in the high load region, the xbind ConnectionManager is able to support a higher throughput with increased latency because it is able to exploit greater parallelism with asynchronous interaction. This is not possible with the sequential implementation.

From Fig. 6, it is clear that the xbind implementation outperforms the other implementations by a very significant margin. The smallest average latency is 11.0 ms for a load of one call/s, and the system can support up to 60 calls/s with average call setup latency of about 40 ms. As a comparison, the maximum call generation rate studied in [10] is 50 calls/s. The performance of xbind ConnectionManager can be significantly improved with the use of message aggregation. The result is shown in Fig. 7, where the message threshold parameter (T) is varied from one to 15.

The tradeoff characteristics of the system between latency and throughput is clear from the figure. For small T, the system
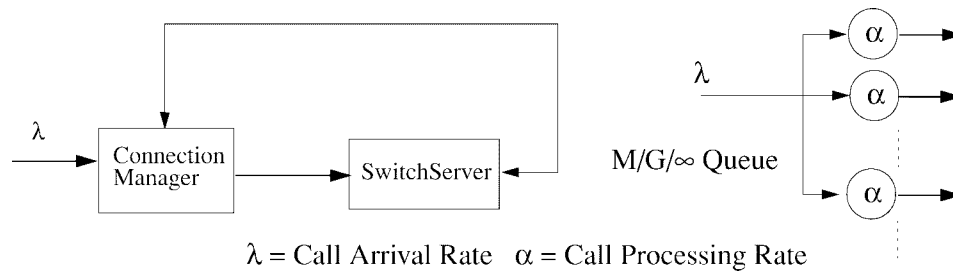
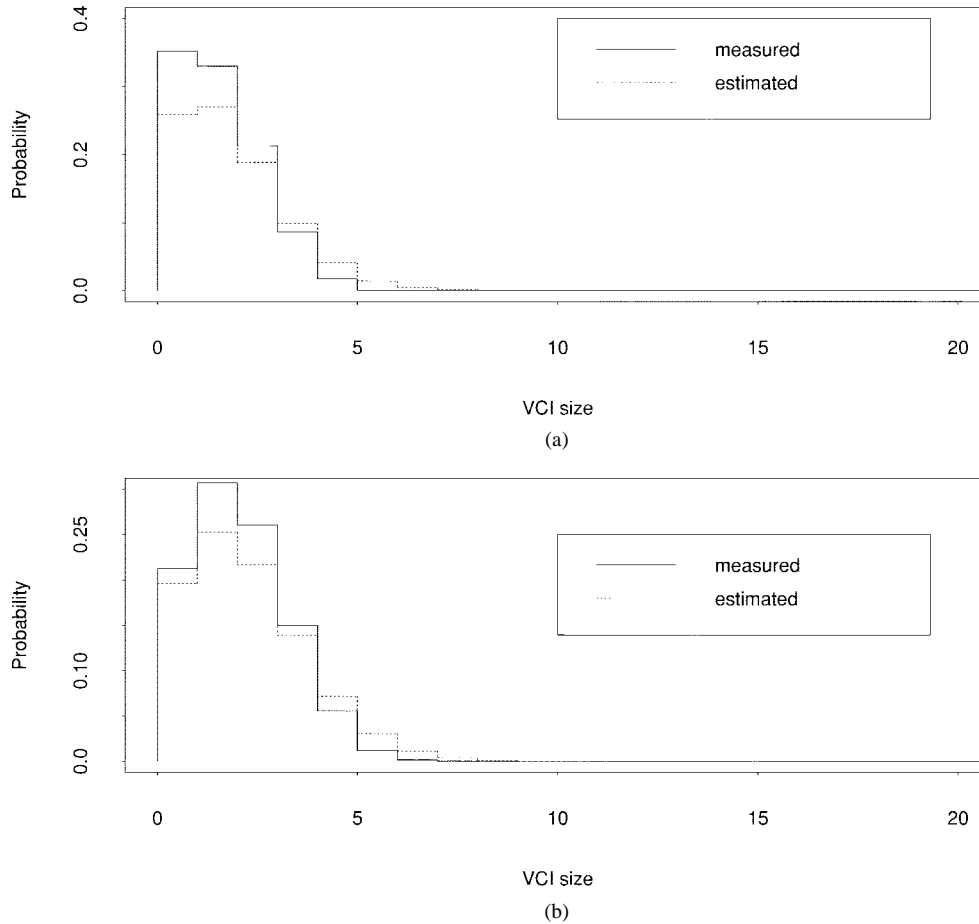Fig. 8. Modeling the processing on the ConnectionManager as a M/G/∞queue.

Fig. 9. Comparison of measured and estimated VCI cache usage distribution (λ = call arrival rate, μ = call holding time, T = message threshold parameter). (a) Low load ($\lambda = 10$, $\mu = 0.2$, $T = 5$). (b) Medium load ($\lambda = 50$, $\mu = 0.2$, $T = 5$).

operates in the low latency mode (<50 ms), but cannot support throughput of more than 100 calls/s. In order to support higher throughput, T must be increased. For larger T, throughput of up to 170 calls/s (600 000 calls/h) is possible, but the average latency has also increased to 85 ms. As a reference point, the 4ESS, a large digital toll switch, has a design objective of supporting 500 000 busy-hour call attempts (BHCA) [15].

A. Dimensioning the Switching Identifier Cache Size

In this section, we discuss how the size of the VPI/VCI cache should be determined for a given call arrival rate. Recall that caching for the switching identifier (a VPI/VCI pair) is performed per output (or input) port per switch. As a result, we will focus the discussion on a single port only. The same methodology can be applied to all output ports.

When a call request arrives and there is a VPI/VCI available in the VPI/VCI cache, the channels on the SwitchServer can be committed in a single phase. This will be referred to as a cache hit. Otherwise, there is a cache miss, and an extra remote invocation is executed to fetch a new VPI/VCI from the SwitchServer. In this section, we identify the size of the VPI/VCI cache required to satisfy the requirement that with probability P, there is a VPI/VCI locally available for immediate use.

The system is modeled in the following way. Assume that there is an infinite number of VPI/VCI available in the VPI/VCI cache, and initially let the variable N_v be zero. When a request arrives, one VPI/VCI is removed from the VPI/VCI cache, and N_v is increased by one. Processing continues, and eventually, a commit channel request is sent
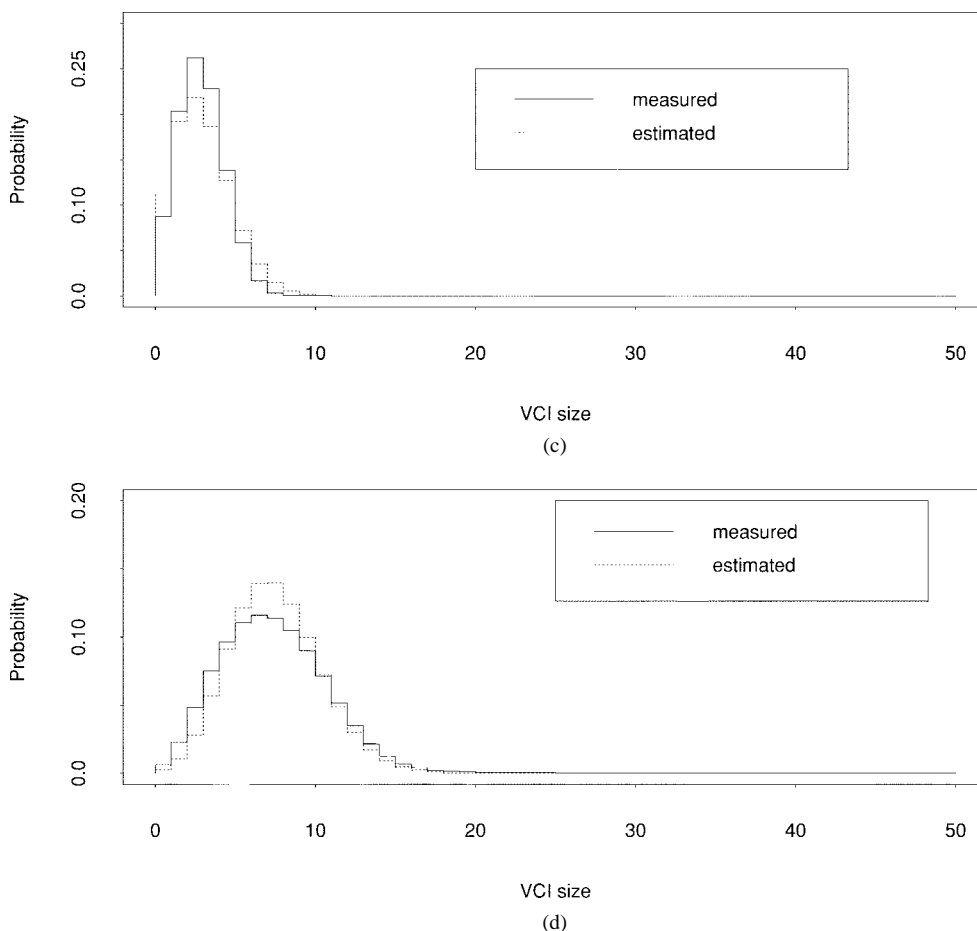
(c)

(d)

Fig. 9. *(Continued.)* Comparison of measured and estimated VCI cache usage distribution (λ = call arrival rate, μ = call holding time, T = message threshold parameter). (c) High load ($\lambda = 100$, $\mu = 0.2$, $T = 5$). (d) High load ($\lambda = 100$, $\mu = 0.2$, $T = 15$).

to the SwitchServer. When the acknowledgment returns from the SwitchServer, the channel has been committed, and N_v is decreased by one. The variable N_v can therefore be interpreted as the number of requests currently being processed, and each of these requests requires a single VPI/VCI.

In the implementation, the number of VPI/VCI pairs is not infinite and has to be replaced. The request for a replacement VPI/VCI is piggybacked onto the commit channel request, and a new VPI/VCI is returned to the cache when the commit channel acknowledgment returns. With such a scheme, the significance of N_v is obvious. N_v is also the number of VPI/VCI's needed to sustain the processing of call setups requested in a single phase. As a result, it is possible to estimate the size of the VPI/VCI cache needed so that a certain percentage P of the total request can be processed in a single phase, if the distribution of N_v, called $P\{N_v \leq X\}$, is known.

In order to have an estimate of the distribution of N_v, we model the system in the following way. The call arrival process is modeled as a Poisson process, and the call processing process time is modeled with a general distribution with average processing rate of α. In addition, we assume that there are an infinite number of servers. Each request is served by a server, and the servers have independent and identical service time distribution. In other words, the system is modeled as a M/G/∞ queue (Fig. 8).

Modeling the system as a M/G/∞ queue allows the distribution of N_v to be computed in a very easy way. In fact, N_v is a Poisson process with mean λ/α [20]. The value of α is the average latency for a VPI/VCI to be replaced or for a call to be setup. It depends on the message threshold parameter, ConnectionManager processing time, communication delay between ConnectionManager and SwitchServer, and the processing time on the SwitchServer. This is intuitively correct as a larger threshold or a larger processing and communication delay will require more VPI/VCI's to be cached for a fixed hit frequency. λ/α can be estimated in two ways, either directly, as the product of the estimated arrival rate and average latency for a VPI/VCI to be replaced, or as the average value of N_V, since the distribution is Poisson. Using this model, we compared the measured distribution of N_v with the estimated distribution (using the average value of N_V measured). This is shown in Fig. 9. The measured distribution is for a total of 100 000 call setups. Note that the region of interest is the region where $P\{N_v \leq X\}$ is large, therefore where $P\{N_v \leq X\} > 0.9$. For these regions, the estimated distributions match the measured distributions rather well.

By modeling the system as a M/G/∞ queue, the question of how to dimension the VPI/VCI cache for each output port can now be answered. An example using two of the plots in Fig. 9 is given in Table I. For each cache size, the estimated

TABLE I
DIFFERENCE BETWEEN ESTIMATED AND MEASURED CACHE HIT FREQUENCY

medium load (λ=50, μ=0.2, T=5)			high load (λ=100,μ=0.2, T=15)		
VCI Cache Size	Estimated Cache Hit Frequency	Measured Cache Hit Frequency	VCI Cache Size	Estimated Cache Hit Frequency	Measured Cache Hit Frequency
5	0.9529	0.9859	12	0.9354	0.9081
7	0.9950	0.9996	13	0.9653	0.9429
9	0.9997	0.9999	15	0.9916	0.9771
10	0.9999	0.9999	20	0.9999	0.9935

and measured cache hit frequencies are shown for comparison. For the medium load case, by keeping ten VPI/VCI's in the connection manager, there is almost no cache miss. For the high load case, 20 VPI/VCI's are sufficient to attain a cache hit ratio of more than 0.99.

As a measure of the overhead incurred by the caching scheme, we use the ratio of N_V to the average number of active calls. For the medium load case, the average number of calls going through the output port is $50/0.2 = 250$ calls. The ratio of VPI/VCI cache size to average call occupancy is therefore $10/250 = 0.04$. For the high load case, the ratio is also 0.04 (20/500). This is a rather small price to pay for a large improvement in throughput-delay characteristics of the call processing system. Another measure of overhead is the storage requirement. This overhead can be estimated in the following way. Let there be 128 nodes in the network, with 32 ports per node. Assume that 32 VPI/VCI's are cached for each port. (This is more than enough to support 100 calls/s with a message threshold of 15, as indicated in Table I.) The total number of VPI/VCI's cached is therefore $128*32*32 = 128K$ VPI/VCI pairs. Let the storage for each VPI/VCI pair be 32 B (we include the overhead to support searching and indexing). The total storage needed is therefore only $128K*32 = 4$ MB, a small requirement given the size of the network and arrival rate considered.

B. Selecting a Message Threshold Parameter (T)

Selecting the optimal values of T is difficult in general because the behavior of the system depends on many factors, and these factors can change dynamically. In this section, we use a simple approach for selecting an appropriate T so that the system can be tuned to support higher throughput given the expected call setup latency.

The problem is simplified by taking into account only one variable, the offered load. Other important variables, like remote invocation latency, are assumed to have minimum variation (excluding local overhead). This simplification is reasonable in an environment where either the ConnentionManagers are the main performance bottlenecks, or the network is partitioned into domains, with one ConnectionManager per domain.

The approach used is based on two observations made from looking at Fig. 7. First, besides the extreme values of T (one and 15 in this case), the throughput-latency characteristics of the system changes gradually with increasing or decreasing value of T. Second, the range of useful T values is not large. To see why this is the case, consider the following example.

Let k be the number of SwitchServers on a path (the number of hops), and let N be the total number of remote invocations. From Fig. 10, we see that $N = 4\lambda(1 + k/T)$. Fig. 10(a) shows a block diagram of one ConnectionManager setting up a connection over three SwitchServers, and Fig. 10(b) shows a plot of how N, the total number of remote invocations changes with increasing value of call arrival rate, λ, for various values of λ using the equation $N = 4\lambda (1 + 3/T)$. Increasing the message threshold parameter T decreases N, but the reduction rate decreases with increasing T. For $T > 10$, the reduction in N is rather insignificant. The asymptotic reduction, and therefore the smallest possible N achieved when $T = \infty$, is shown for comparison purposes. The incremental decrease in N becomes negligible when k/T becomes much larger than one. In this case, the gain in reduction in remote invocation is likely to be outweigh by increase in latency.

Based on these observations, our approach is to use a static threshold scheme for changing the parameter T, based on the observed traffic load. Since the changes in terms of throughput and latency is gradual, it is not necessary to tune T precisely. Instead, the operation of the connection manager can, in general, be classified into a number of regions. For the system implemented, three regions are sufficient. A low traffic load region (0–50 call/s), a medium traffic load region (50–100 call setup/s), and a high traffic load region (>100 call setup/s). A threshold of one is used for the low traffic case, which provides low latency call setup, but only up to 50 calls/s. A threshold of four (or five) is used for the medium load case. For arrival rates of greater than 100 calls/s, a threshold of ten is used. By estimating the call arrival rate, the connection manager can dynamically change its threshold parameter so as to obtain a good tradeoff between latency and throughput. A reproduction of Fig. 7 with only three threshold parameters is shown in Fig. 11.

V. CONCLUSION

In this paper, we described a call processing system for ATM switching system that runs on a CORBA-based distributed processing environment. The system has low latency and high throughput, demonstrating that good performance can be achieved using a general purpose distributed processing plat-
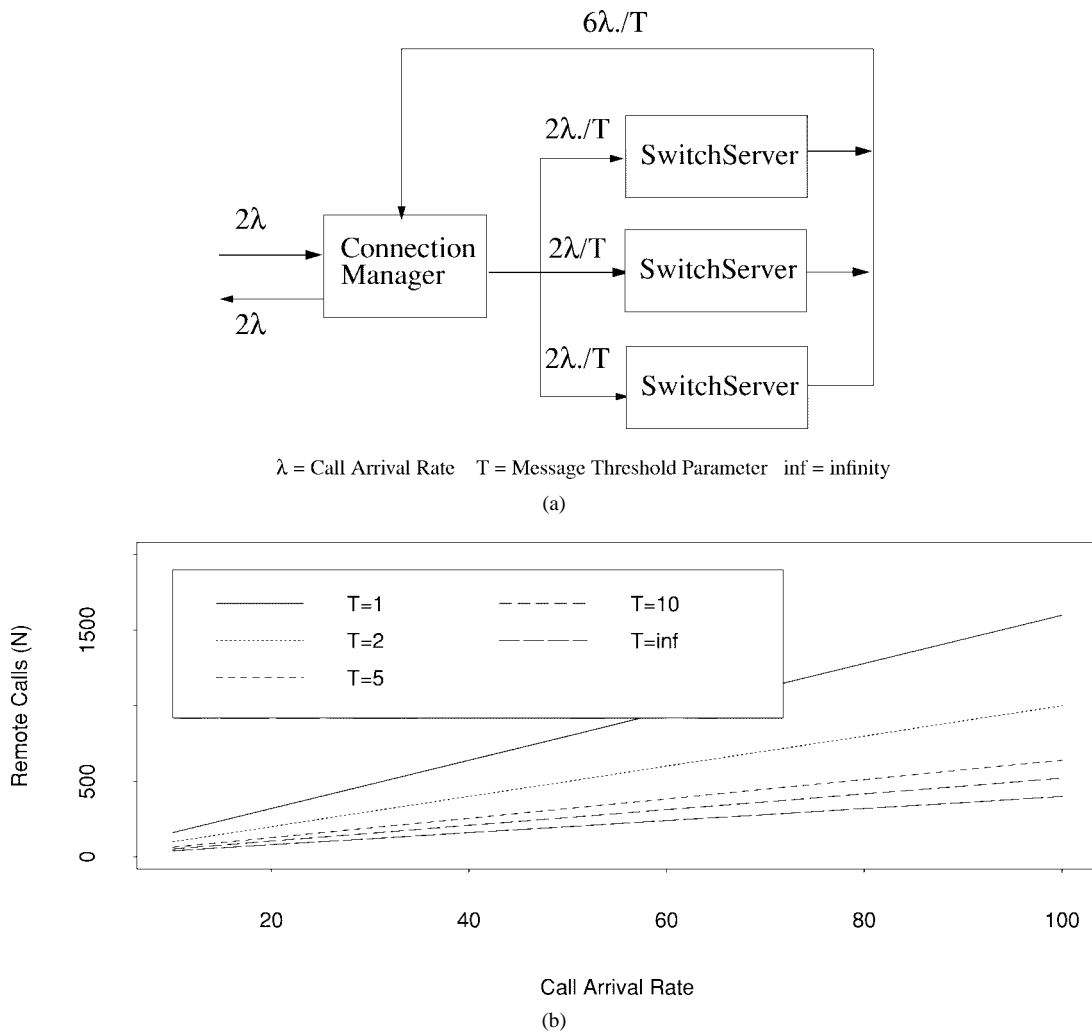
Fig. 10. Remote invocation load on the ConnectionManager.
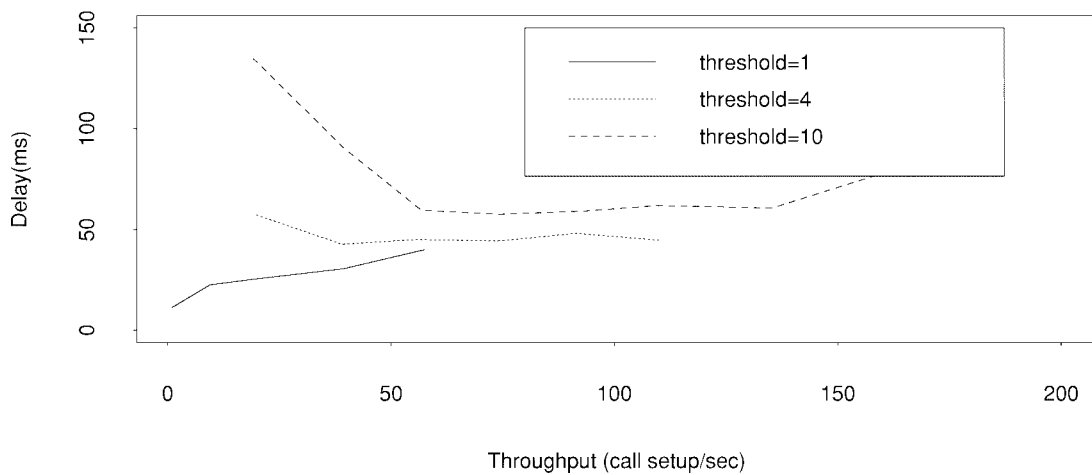

Fig. 11. Throughput-delay characteristics of xbind connection manager with $T = 1, 4, 10$.

form for network signaling. We have demonstrated how different techniques can be used to improve signaling performance. The performance figure presented is specific to the experimental setup used. We observed that use of faster machines (or RPC mechanisms) will invariably improve the performance. By using Sun Ultra2 instead of the older HP9000/700, the minimum latency improves to about 8 ms, and the highest throughput supported increases to about 190 calls/s.

The components of the connection manager consist of a connection manager state machine, a message aggregation queue, a route cache module, a name (VPI/VCI) cache module, and a bandwidth cache module. The latency and throughput of call processing is improved by caching and message aggregation schemes that reduce the number of remote accesses. Parallel processing of a single call request is also used to enhance the performance of call processing. The behavior of the throughput-delay characteristics of the call processing system was studied with respect to two control parameters, namely: 1) message aggregation threshold and 2) size of switching identifier cache. These control parameter values can be tuned to achieve the desired tradeoff among resource utilization, call setup latency, and call throughput.

ACKNOWLEDGMENT

REFERENCES

[1] A. Battou, "Connection establishment latency: Measured results," *ATM Forum Document*, ATM-Forum/96-1472, Oct. 1996.
[2] M. C. Chan, J. K. Huard, A. A. Lazar, and K.-S. Lim, "On realizing a broadband kernel for multimedia networks," in *Proc. 3rd COST 237 Int. Workshop Multimedia Telecommunications and Applications*, Barcelona, Spain, 1996, pp. 65–74.
[3] A. Gokhale and D. Schmidt, "Techniques for optimizing CORBA middleware for distributed embedded systems," in *Proc. INFOCOM'99*, New York, NY, pp. 513–521.
[4] J. Huard and A. A. Lazar, "On QOS mapping in multimedia networks," in *Proc. 21st IEEE Ann. Int. Computer Software and Application Conf. (COMPSAC'97),* Washington, DC, pp. 312–317.
[5] A. A. Lazar, K. Lim, and F. Marconcini, "Realizing a foundation for programmability of ATM networks with the binding architecture," *IEEE J. Select. Areas Commun.*, vol. 14, pp. 1214–1227, Sept. 1996.
[6] K. van der Merwe and I. Leslie, "Switchlets and dynamic virtual ATM networks," in *Proc. IFIP/IEEE Int. Symp. Integrated Network Management (IM'97),* San Diego, CA, pp. 355–368.
[7] K. Murakami, R. Buskens, R. Ramjee, Y. Lin, and T. La Porta, "Design, implementation, and evaluation of highly available distributed call processing systems," in *Proc. FTCS'98*, pp. 118–127
[8] J. Nagle, "Congestion control in IP/TCP internetworks," in Network Working Group RFC 896, Jan. 1984.
[9] P. Newman, R. Hinden, E. Hoffman, F. C. Liaw, T. Lyon, and G. Minshall, "General switch management protocol specification—Version 1," Palo Alto, CA, Mar. 1996.
[10] D. Niehaus, A. Battou, A. McFarland, B. Decina, H. Dardy, V. Sirkay, and B. Edwards, "Performance benchmarking of signaling in ATM networks," *IEEE Commun. Mag.*, vol. 35, pp. 134–143, Aug. 1997.
[11] H. Oliver, S. Brandt, A. Thomas, and N. Charton, "Network control as a distributed object application," *Distributed Systems Engineering*, vol. 5, pp. 19–28, Mar. 1998.
[12] S. Ohta and K. Sato, "Dynamic bandwidth control of the virtual path in an asynchronous transfer mode network," *IEEE Trans. Commun. Technol.,* vol. 40, no. 7, pp. 1239–1247.
[13] A. Orda, G. Pacifici, and D. E. Pendarakis, "An adaptive virtual path allocation policy for broadband networks," in *Proc. INFOCOM'96*, San Francisco, CA, pp. 1285–1293.
[14] N. G. Aneroussis and A. A. Lazar, "Virtual path control for ATM networks with call level quality of service guarantees," *IEEE/ACM Trans. Networking*, vol. 6, pp. 222–236, Apr. 1998.
[15] M. Schwartz, *Telecommunication Networks: Protocols, Modeling and Analysis.* Reading, MA: Addison-Wesley, 1987.
[16] I. T. Ming-Chit, W. Wang, and A. A. Lazar, "A comparative study of connection setup on a connection management platform," in *Proc. First IEEE Conf. Open Architectures and Network Programming*, San Francisco, CA, 1998, pp. 14–24.
[17] TINA-C, *Service Architecture Version 2.0,* Document TB_MDC.012_2.0_94, Mar. 1995.
[18] M. Veeraraghavan, G. L. Choudhury, and M. Kshirsagar, "Implementation and analysis of PCC (parallel connection control)," in *Proc. INFOCOM'97,* Kobe, Japan, 1997, pp. 833–842.
[19] M. Veeraraghavan, T. F. La Porta, and W. S. Lai, "An alternative approach to call/congestion control in broadband switching systems," *IEEE Commun. Mag.*, vol. 33, pp. 90–97, Nov. 1995.
[20] R. W. Wolff, *Stochastic Modeling and the Theory of Queues.* Englewood Cliffs, NJ: Prentice-Hall, 1989.

Mun Choon Chan (M'97) received the B.S. degree from Purdue University, West Lafayette, IN, in 1990 and the M.S. and Ph.D. degrees from Columbia University, NY, in 1993 and 1997, respectively, all in electrical engineering.

From 1991 to 1997, he was a member of the COMET Research Group, working on ATM control and management. Since 1997, he has been a Member of the Technical Staff at Bell Laboratories, Lucent Technologies, Holmdel, NJ. His current interests include wireless data networking and network management.

He is a member of the ACM, and he serves on the Technical Program Committee of IEEE INFOCOM'2000.

Aurel A. Lazar (S'77–M'80–SM–90–F'93) has been a Professor of Electrical Engineering at Columbia University, New York, NY, since 1988. His current theoretical research interests are on networking games and pricing. His experimental work, which focuses on building open programmable networks, has led to the establishment of the IEEE Standards working group on Programming Interfaces for Network. He was instrumental in establishing the OPENSIG international working group with the goal of exploring network programmability and next generation signaling technology. Currently on leave from Columbia University, he is Chairman and CEO of Xbind, Inc., a high technology startup company in New York, NY.

Dr. Lazar was the Program Chair of the Fall and Spring OPENSIG'96 workshops and of the First IEEE Conference on Open Architectures and Network Programming.