NASA-CR-197768

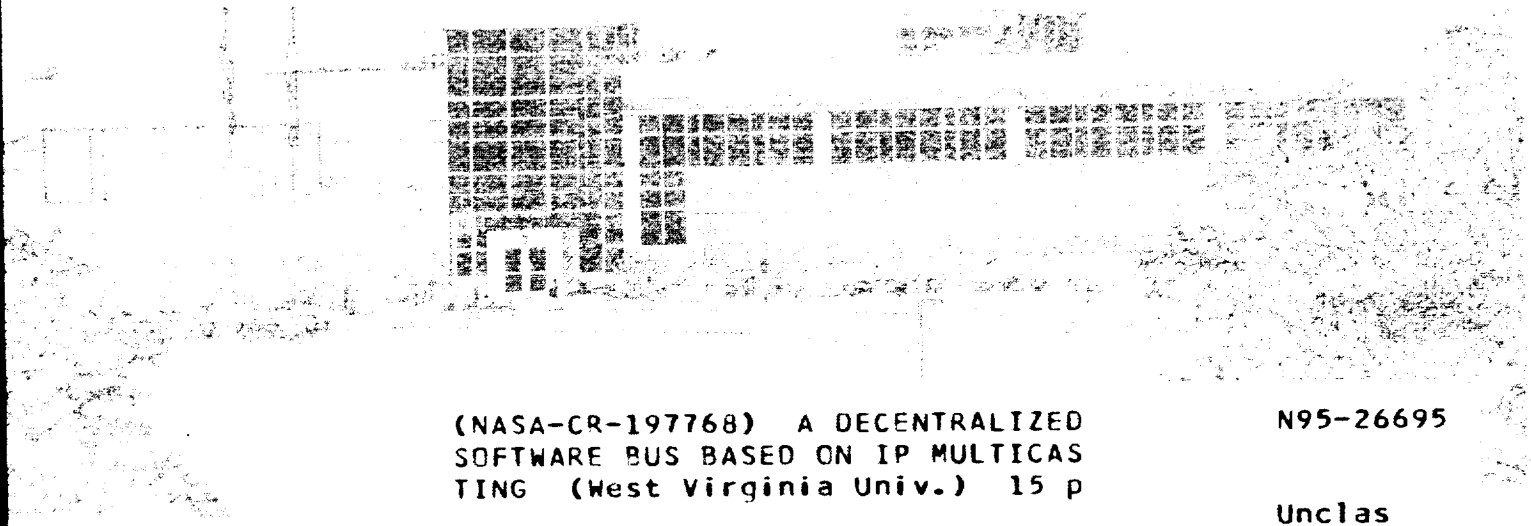
NASA/WVU Software IV & V Facility
Software Research Laboratory
Technical Report Series

NASA-IVV-94-001
WVU-SRL-94-001
WVU-SCS-TR-95-2
CERC-TR-TM-94-012

IN-61-CR
48516
P-15

A Decentralized Software Bus based on IP Multicasting

by John R. Callahan and Todd Montgomery



National Aeronautics and Space Administration

West Virginia University

[illegible]	4-10-95	[illegible]	4-19-95
George J. Sabolish	Date	John R. Callahan	Date
Manager, Software Engineering		WVU Principal Investigator	

A Decentralized Software Bus based on IP Multicasting

John R. Callahan[1]
Todd Montgomery
Department of Statistics & Computer Science
Concurrent Engineering Research Center
West Virginia University

Abstract

We describe a decentralized reconfigurable implementation of a conference management system based on the low-level Internet Protocol (IP) multicasting protocol. IP multicasting allows low-cost, world-wide, two-way transmission of data between large numbers of conferencing participants through the Multicasting Backbone (MBone). Each conference is structured as a *software bus* --- a messaging system that provides a run-time interconnection model that acts as a separate agent (i.e., the bus) for routing, queuing, and delivering messages between distributed programs. Unlike the client-server interconnection model, the software bus model provides a level of indirection that enhances the flexibility and reconfigurability of a distributed system. Current software bus implementations like POLYLITH [1], however, rely on a centralized bus process and point-to-point protocols (i.e., TCP/IP) to route, queue, and deliver messages. We implement a software bus called the MULTIBUS that relies on a separate process only for routing and uses a reliable IP multicasting protocol for delivery of messages. The use of multicasting means that interconnections are independent of IP machine addresses. This approach allows reconfiguration of bus participants during system execution without notifying other participants of new IP addresses. The use of IP multicasting also permits an economy of scale in the number of participants. We describe the MULTIBUS protocol elements and show how our implementation performs better than centralized bus implementations.

1 Introduction

Most distributed systems are comprised of programs that must identify and locate one another in order to exchange messages and carry out a computation. Some of these programs run continuously for long periods of time (e.g., servers) while others are more transient (e.g., clients). In addition, most of these programs are under autonomous control of separate users and other programs. Often, we cannot be certain if or where a program is executing in a distributed system. The uncertainty of locating other programs in a distributed system makes distributed programming complex and prone to faults.

The client-server model [2] attempts to make distributed programming less complex by associating logical names with remote services. Given the logical name of a service, a client relies on a name server to locate the IP address of the required service. This approach has proved adequate in practice but suffers from several problems. First, the name server is another program that needs to be located usually at some well-known address. Second, if any program changes its location, it must notify the name server. Third, any client program must rely on the name service for each remote invocation because past remote addresses may not be up-to-date. Finally, if each client-server connection is implemented via a point-to-point protocol (e.g., TCP/IP) then it is not possible to scale the system to large numbers of clients. These

[1]This work is supported by DARPA Grant MDA 972-91-J-102 under the DARPA Initiative for Concurrent Engineering (DICE) program, the National Library of Medicine (NLM), and NASA Grant NAG 5-2129 under the Independent Software Verification and Validation (IV&V) program.

problems and limitations are the result of the primary weakness of client-server model that requires each program to be responsible for routing and delivering messages directly to the intended recipients.

To eliminate the need for direct message routing and delivery in the client-server model, the software bus model [1] introduces a separate agent responsible for routing, queuing, and delivery of messages. Interconnections between programs are specified separately. Every software bus consists of one or more bus slots and a bus process that delivers messages between slots. Individual programs register in each slot of the bus and interact only via the slot interface. The bus process acts as a message delivery service between programs in slots. When a slot produces a message, it will be delivered according to the current interconnection topology. If a message is destined for an unoccupied slot, the message may be dropped or queued until the slot is occupied at some point in the future. The interconnection map maintained by the bus process may be changed during execution in order to reconfigure the system. While this is done more easily in a software bus implementation than a client-server implementation like remote procedure call (RPC) [3], it is still a complex operation especially if there are queued messages.

While the software bus provides a level of indirection needed in many distributed programs, the overhead of a separate bus process to implement this abstraction is costly. The bus process becomes a bottleneck because all messages must be routed through it. In current implementations using a centralized bus process, messages must be routed through the bus process and then to each recipient individually via point-to-point protocols.

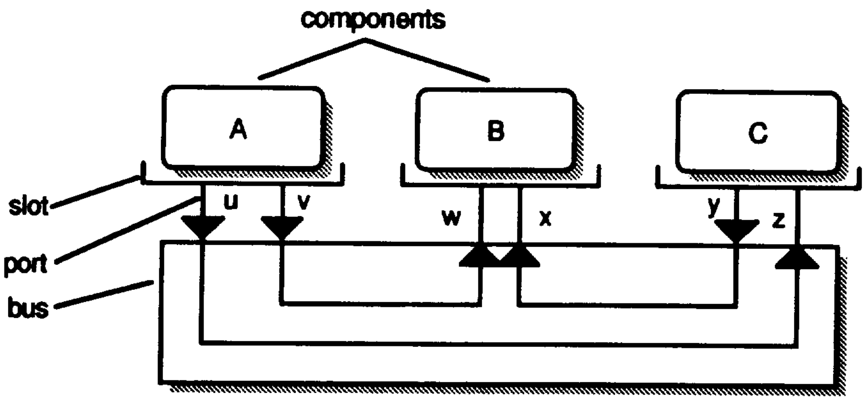
We have implemented a software bus called the MULTIBUS that provides the flexibility of the software bus abstract model, but provides increased fault tolerance, reconfigurability, and performance because we eliminate the bottleneck of a centralized bus process. The MULTIBUS uses a separate process, called a *router process* to direct messages to their destinations, but relies on a IP multicasting to deliver messages simultaneously between multiple recipients. Each bus participant needs only the group multicast address to send and receive from other bus participants. This approach increases flexibility and performance over centralized bus implementations and can be reconfigured more easily than either the centralized bus or client-server models because each bus participant need not know the exact IP address of any other participant.

1.1 Software Bus

The software bus model allows programs that are part of a larger system to be independent of the communications topology of that system. The model assumes that each slot in the bus is occupied by a software module that acts as a "black-box" with multiple ports. Figure 1 illustrates a simple software bus configuration. Messages produced on ports are delivered to ports of other slots in the bus. If a slot is unoccupied or otherwise busy, the message is queued for delivery until the recipient requests the next message in the queue.

The bus interconnection topology is specified separately in a module interconnection language (MIL) [4]. The specification serves as the initial topology or a static topology depending on the needs of the application. A MIL specification determines the module interconnection graph for a distributed application. This graph determines the routing of messages between ports of different modules and the characteristics of the interconnections themselves.

The modular structure of the bus is centered around a backplane across which components exchange messages. This design promotes an open systems architecture making it easier for third party vendors to "plug-in" their tools. The bus model makes future enhancements simpler than in more monolithic designs. It improves over the client-server model by adding a level of indirection between programs in a distributed system. Client and server programs need not know about the run-time details of each other in order to interact.



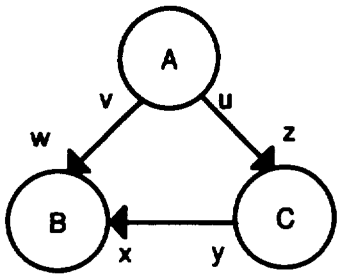


Figure 1: A software bus and its associated module interconnection graph.

Even with its flexible message passing scheme, the software bus abstraction is inadequate for distributed systems comprised of indeterminate numbers of transient programs. We have extended the abstract software bus model to accommodate transient clients and indeterminate numbers of slot participants. These enhancements include slot types, occupancy limits, connector and port types as described below. Such enhancement were necessary to accommodate the types of transactions between long-term and short-term applications that occupy different bus slots at different times during system execution.

Slot Type. A slot may be **required**, **transient**, or **invoked**. A required slot must be occupied at all times during the execution of the distributed system. The router process, for instance, occupies a required slot. A transient slot may be unoccupied at some time. An invoked slot implies that only when a message is delivered to the slot is a program invoked. The last type of slot is particularly useful in applications where messages delivered to a bus slot are simply concatenated to a log file by an invoked script.

Slot Occupancy. Two occupancy numbers are associated with each slot that determine the minimum and maximum number of enrolled programs per slot. A slot with minimum occupancy of one is a required slot while a minimum of zero implies that the slot is transient or invoked.

Connector Types. The behavior of a message sent between ports is determined by the characteristics of the connection between them. Current bus implementations queue delivery of messages on all ports, but we enhance the definition of a connection as having a finite capacity and message time-to-live (TTL) value. If a message is queued in a connection for more than its time-to-live, then it is dropped.

Port Types. Messages from one port to another may need to retain information about their origin as in the case of request-response transactions particularly for slot with multiple occupants. For instance, the response to a request from a single client in a multiple occupancy slot must be returned to that client. A port type in the message will ensure the response delivery to the sender. Other port types may define asynchronous messages that do not need information about the originator.

These enhancements to the original software bus model are necessary to accommodate the reconfiguration needs of distributed programs that interact via a bus model but are under autonomous control of individual users and other programs. Such programs include group conferencing tools, and database servers. These tools may run continuously as in required bus slots, while transient clients connect via transient slots with multiple occupancies. The bus defines the types of transactions permitted between tools engaged in these roles.

1.2 IP Multicasting

Many software bus implementations support message passing through a centralized mechanism. This added level of indirection, however, is costly in terms of performance. The cost is high because a separate process must route, queue, and deliver all messages creating a bottleneck in the distributed system. We introduce a technique based on a IP multicasting that does not require a centralized delivery mechanism. We have implemented a software bus using a modified version of the VMTP protocol [5] for reliable multicasting in distributed environments.

IP multicasting is a transport level protocol similar to point-to-point IP. Unlike point-to-point packets that are delivered between two distinct Internet hosts, multicast messages are selectively broadcast on local subnets and received by any host that claims to be a member of a multicast address group. Like point-to-point IP, applications transmit and listen on bound BSD sockets for messages. The only difference is that messages are delivered to groups of logical hosts rather than a specific host.

Software buses based on broadcast IP have been implemented, but multicasting is not broadcasting. A multicasting IP message is received only by those hosts claiming to be a member of a virtual host group on a given port. A single machine may be part of several host groups and on different ports. IP multicast messages are not forwarded by standard routers, but they can be propagated world-wide over the existing Internet network via a network of multicast routers that constitute the Multicast Backbone (MBone). The multicast routers use IP tunnels to forward multicast message between sub-networks. Today, there are several hundred of these routers operational. In the future, commercial routers will support forwarding of multicasting traffic.

Multicasting is highly useful in distributed systems that consist of transient components that need to be reconfigured frequently. There is no need to keep track of host addresses when an application moves between host machines. The newly configured application can reconnect with the system by joining the IP multicast group. This flexibility makes multicasting useful in local and wide area distributed environments where the number of bus participants may be unknown as well as their geographical distribution.

A software bus model based on multicasting as in the MULTIBUS offers a high degree of flexibility as well as structure to distributed systems. Many existing multicast applications such as audio and video teleconferencing [6] use unreliable datagram messages that permit eavesdropping and unstructured transactions between applications. Our approach places some structure on the types of transactions between bus applications but allows for flexibility in reconfiguring the system in the face of changes or faults.

Many existing distributed systems that rely on remote procedure call [7] [8] or special point-to-point messaging are not easily reconfigurable because the failure of a single element may cause the system to enter an undefined state. For example, CONIC [9] relies on operating system level primitives while others like PVM [10] use messaging servers on each host to route and deliver messages. Conic executes on a single host where failures of single elements imply system failures. In PVM, the absence of a pvmd (a message daemon process) will cause the application to fail. The original POLYLITH system relies on a single bus process at a well-known host address and port to act as a message clearinghouse on a local or

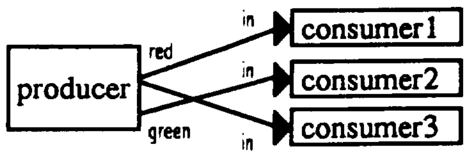


Figure 2: Module interconnection graph for producer-consumers example

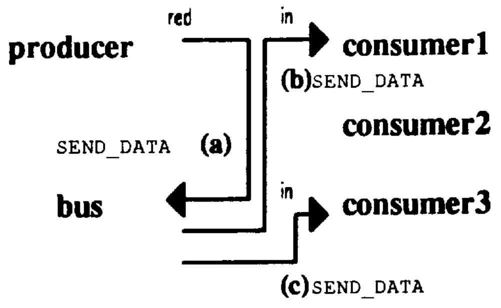


Figure 3: Messages needed to send data in a centralized bus implementation

wide-area network. While failure of the POLYLITH bus process means failure of the system, its true problem is that it creates a bottleneck in message delivery. The MULTIBUS is an improvement over these solutions because its reliance on IP multicasting allows resiliency in the face of transient failures and simple dynamic relocation of computing resources.

2 Implementation

The topology of a module interconnection graph is independent of how the modules and connections are implemented by any software bus. Figure 2 shows a module interconnection graph for a single producer and three consumers. The producer's red port is connected directly to the input ports of two consumers. The producer's green port is connected to only one consumer. A message sent on the red port is multiplexed to the appropriate consumers by the software bus. The bus routes all messages via three different connections labeled

```
producer'red   ◊ consumer1'in
producer'green ◊ consumer2'in
producer'red   ◊ consumer3'in
```

In a centralized software bus implementation, at least three messages (not counting acknowledgments) are required to send the message to the intended recipients. Figure 3 depicts a centralized bus implementation of the producer-consumer example. First, a SEND_DATA message is sent on the producer's red port to the bus process where it is copied, routed, and queued for delivery to each recipient's input port (a). The bus then sends the message to each consumer who receives a SEND_DATA message after being interrupted by the bus process or by polling the bus process for new messages (b & c). The centralized bus requires at least one message for each recipient because it uses point-to-point protocols like TCP/IP [11] for reliable delivery.

Figures 4 and 5 depict messages in an MULTIBUS multicast bus implementation for the producer-consumers example. In Figure 4, a single multicast SEND_DATA message is sent by the producer and received by all bus slots including the router process (a) but each receiver must first determine or not to

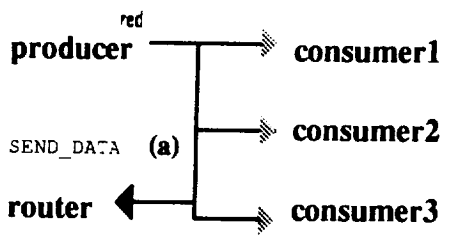


Figure 4: A multicast data message in the Multibus

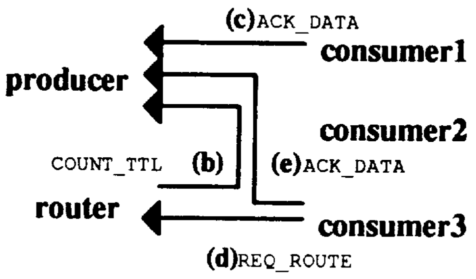


Figure 5: Acknowledgment and routing messages in the Multibus

accept this message (shown as grey arrows) by referring to the router process. In this example, we assume that consumers 1 and 2 have already "cached" routing information, but consumer 3 has no information. The responses to the multicast SEND_DATA message are shown in Figure 5. The router process responds to the message by returning a COUNT_TTL message with count of the number of expected receivers who should acknowledge the producer's message based on the router's connection table (b). In this case, the COUNT_TTL should be equal to two because there should be only two acknowledgments. In parallel with the router's acknowledgment count response, each accepting consumer sends an ACK_DATA response to the producer in acknowledgment of its acceptance of the data (c). The producer retransmits the message (after an appropriate time-out period) until all expected acknowledgments are received. The VMTP protocol guarantees that retransmissions will be dropped by sites that have already received the multicast message.

In Figure 5, consumer 1 returns an ACK_DATA message immediately while consumer 3 first queries the router process. This is because consumers 1 & 2 already known whether or not to accept the message, but consumer 3 must query the router process for the current topology. The router's response message (not shown) is multicast so that all bus participants can see the routing information and cache it if necessary.

There are several special issues related to the operation of the multicast bus. We must ensure that messages are routed properly. The bus must ensure that messages are queued for asynchronous delivery. The bus must support dynamic reconfiguration of the interconnection topology. We now discuss the implementation of mechanisms designed to deal with these issues in the multicast bus implementation.

2.1 Routing

A multicast message will be *accepted* by a bus slot if that slot owns the message destination port(s). Otherwise, the message is unacceptable. Each slot determines whether or not a message is acceptable via local routing tables or by querying the router process with a REQ_ROUTE message (d). A router query message is multicast because the router may dynamically change location during system execution. The router process responds to route requests (e) with an ANS_ROUTE message that tells the bus slot whether or not it is a destination and the destination port. After determining the correct routing for a message and accepting it, the consumer then responds to the producer's original data message.

Each slot may cache this information in a local routing table for future use. When the routing is dynamically reconfigured, the router process will send a multicast message to all bus slots with the new information. It is up to each slot to either invalidate its entries in its cache or update its local routing table. Since each bus slot may maintain local tables, messages may be routed at their source or destination. The router process is the final arbiter of routing. For source routing, the source may specify destination ports in a SEND_DATA message. For destination routing, the sender leaves the destination(s) argument of a SEND_DATA message undefined. While each slot can specify the intended destination(s) in a SEND_DATA message, the receiver can choose to ignore this information. A receiver can rely on REQ_ROUTE messages and ANS_ROUTE responses to ensure correct delivery.

In static configurations, however, it may be useful to use local routing. Initially, slots will use REQ_ROUTE messages to determine destinations, but once the tables are initialized, then each slot need not rely on the router. This case is analogous to the direct connection implementation scheme proposed by POLYLITH as an alternative to a bus process. Our scheme has comparable performance with much simpler reconfiguration properties.

2.2 Queuing

Message queuing is done by the sender. Queuing is based on finite queue capacities and message time-to-live limits. When a bus slots sends a message and does not receive sufficient acknowledgments within a time-out period, it will retransmit the message until all acknowledgments are received or a longer time-out is reached. The VMTP `invoke` function returns after the first response is received. Subsequent responses are received via the `getreply` function. We have modified the VMTP library so that if all acknowledgments to a message are not received by the sender, then the message is queued. The sender will interrupt itself periodically and retransmit or flush any queued messages. Messages are flushed when their time-to-live (TTL) has expired. If the process message queue is full, then the call to the modified `invoke` will return an error status.

2.3 Reconfiguration

Execution of a new bus begins by starting and initializing the router process with an interconnection map. This map may be altered during system execution by ADD_ROUTE and DEL_ROUTE messages from authorized bus occupants. As an option, a bus can be configured with an authorized *shadow slot* through which monitoring and reconfiguration tools can also enroll and make changes to the interconnection map.

Another form of reconfiguration involves the exiting of a bus occupant from a slot and the initialization of a new bus occupant. If a bus occupant decides to terminate and restart execution at another location, it needs only to send a DROPOUT message to the router process. No other bus participants need to be involved in the transaction. Messages will queue up at their source for the process during the reconfiguration period. Once the restarted process is enrolled with the router process, it can immediately receive the queued multicast SEND_DATA messages.

One of the major advantages to this approach is that any process can be reconfigured, including the router process, without direct knowledge of the physical IP address of any other bus participant. All bus participants locate communicate via a logical IP multicasting address. This eliminates the need to update addresses and port numbers in other applications. While the POLYLITH software bus implementation also enables this type of flexibility, it does so only for bus participants, not the bus process and assumes that no connections are direct between bus participants. The MULTIBUS allows flexibility for all bus participants without the overhead of a central message delivery mechanism.

Request Message	Message Description	Possible Responses
ENROLL	request to occupy bus slot	TICKET, NEG_ACK
DROPOUT	vacate bus slot	DROPPED, NEG_ACK
KEEP_ALIVE	ping message to slot occupants	ALIVE
REQ_DATA	data message	COUNT_TTL, ACK_DATA
REQ_ROUTE	request routing information	ANS_ROUTE
ADD_ROUTE	add a connection	ACK_ROUTE, NEG_ACK
DEL_ROUTE	delete a connection	ACK_ROUTE, NEG_ACK

Table 1: Multibus request messages

2.4 Protocol

This section describes the protocol messages exchanged between bus slots and the router process in a multicast bus implementation. All messages between bus slots and the router are multicast for dynamic reconfiguration purposes. If the router process changes location during the execution, it can reconnect to the bus conference without reconfiguring any static point-to-point tables. Indeed, bus conferences may occur over wide area networks with bus slots moving from site to site.

2.4.1 Requests Messages

Request messages are multicast to the entire software bus, received by all bus slots, and accepted only by intended receivers according to the current routing. We rely on the VMTP protocol to ensure reliable multicast delivery. In accordance with the VMTP protocol, multicast requests are followed typically by multiple, point-to-point responses. Table 1 lists the request protocol elements and their associated response messages.

The ENROLL message is issued by an application that wishes to occupy a bus slot. The ENROLL message is multicast so that the current router process can be found anywhere within the scope of the IP multicast. If successful, a ticket is returned by a point-to-point response message from the router process to the new slot occupant. If the slot is already occupied up to its occupancy limit or no such slot is available, then a negative acknowledgment is returned. The ENROLL message is sent reliably in the sense that the router process must return either a ticket or a negative acknowledgment.

This DROPOUT message is used to terminate occupancy of a slot by an application. The DROPOUT message can be denied by the router process for some reason but must eventually be honored. Before a DROPPED message is issued in response, the router process will issue DEL_ROUTE messages (see below) to eliminate dependent interconnections and reconfigure the application before allowing the occupancy to terminate.

The router process periodically checks on the liveness of slot occupants by "pinging" them with KEEP_ALIVE. If there is no response after an appropriate number of retries, then the slot occupancy is decremented and the ticket count increased. If the occupant returns with an old ticket, it will not be able to send data to other participants. The SEND_DATA message is multicast by a slot in order to send data on a particular port. All bus slots receive the message and use either their static routing tables or rely on the router process to determine whether or not to accept the message. In response to a SEND_DATA message, the router process returns a COUNT_TTL message with the expected number of acknowledgments and a time-to-live (TTL) for the message based on the longest TTL of any interconnection attached to the data port on which the message was sent. Other bus participants response with ACK_DATA messages to acknowledge receipt and acceptance of the message.

Response Message	Message Description
ALIVE	acknowledge slot occupancy
TICKET	returned by enrolling
DROPPED	positive dropout acknowledgment
COUNT_TTL	acknowledgment count and time-to-live for data message
ACK_DATA	acknowledge acceptance of data message
ANS_ROUTE	routing information about a data message
ACK_ROUTE	returned when adding or deleting a connection
NEG_ACK	any negative acknowledgment

Table 2: Multibus response messages

A REQ_ROUTE message is issued by a bus occupant if it needs to know how to route a SEND_DATA message to one of its ports. The request is multicast so that the router process can be found dynamically and the response is directed specifically to the requester. The response (a ANS_ROUTE message) helps the bus occupant determine whether or not to accept the message and on which port.

The ADD_ROUTE and DEL_ROUTE messages can be issued by any authorized bus occupant to add or delete an interconnection respectively. The router process is the only intended recipient of the message, but it is multicast in order that the bus occupant need not know where the router process is located.

2.4.2 Response Messages

Responses are point-to-point messages issued in reply to a request message (see above). The VMTP protocol provides a reliable, multicasting request-response capability where all recipients are required to reply with a point-to-point response. Table 2 lists the types of response messages.

An ALIVE message is sent in response to a KEEP_ALIVE message. Each slot occupant must reply to a KEEP_ALIVE message or else its ticket will expire. A bus occupant acquires a TICKET message from the router process by sending an ENROLL multicast message. Any slot occupant can leave the bus by sending a DROPOUT message and getting a DROPPED in response.

A COUNT_TTL message is sent by the router process in response to a SEND_DATA message. The COUNT_TTL message contains the number of expected ACK_DATA messages the sender should receive from accepting receivers. The COUNT_TTL message also contains the time-to-live (TTL) for the message. The TTL is the length of time the message should spend in the queue being retransmitted before it is flushed. The sender of a SEND_DATA message can receive ACK_DATA messages immediately after sending and any time before the COUNT_TTL message from the router process but the send will not proceed until the COUNT_TTL is received.

A ANS_ROUTE message is sent by the router in response to a REQ_ROUTE message. It contains a Boolean and a list of destination ports. If the Boolean is true, the list is used to accept the message on specific ports.

2.4.3 Information Messages

Information message are multicast message directed to all slot occupants of a bus. They are used to maintain local routing tables if used by slot occupant processes. Each slot must acknowledge the message

Information Message	Message Description
ROUTE_ADDED	a connection has been added
ROUTE_DELETED	a connection has been deleted

Table 3: Multibus information messages

but it may decide to discard the message or use it to maintain its local routing table. Table 3 lists the information messages in the MULTIBUS.

Both messages are sent by the router process in response to the addition or deletion of an interconnection. It must be acknowledged by all enrolled bus slots. Its contents can be ignored by bus occupants that do not maintain their own routing tables. Multicast updates of changes to the router table can be disabled as an option when the bus is started.

3 Discussion

One of our major goals in designing and implementing the multicasting software bus was to establish a new means of interaction between distributed programs instead to the client-server model. We needed a method of handling transient and decentralized reconfigurations without interruption of service. The Multibus meets these goals, but we also needed a means of monitoring conferences and making them globally visible list without users knowing details of the bus infrastructure. Since a multicasting conference is a virtual service[2], unlike conventional services where applications connect to well-known "sockets" on designated servers, we needed a way to list those available buses by listening to the network for periodic multicast messages. Each executing router process of a bus periodically sends an unreliable datagram advertising its presence. These datagrams include the multicast address of the bus through which a user can join a bus conference.

A bus conference is a collection of bus slots (and their possible occupants) that are currently available to join by starting a new application in an open bus slot. From previous experience with message services like PVM, we found that when an application is comprised of multiple services that are interdependent on one another, it becomes difficult to coordinate their execution. For example, if a server fails, it is difficult to isolate which one failed. The software bus model solved some of these problems, but users were still unaware of how they fit into a conference, what tools they needed, and how to join the application. To solve the problem of global visibility, we created a tool called the Communications Manager (CM) that maintains a list of buses in a system without relying on a central registry by listening for periodic bus advertisements. Figure 6 shows a picture of the CM window tool that lists buses currently available on the network. It lists all currently active buses on a network and the status of their slots. A bus is removed from the list if the CM tool receives a cancellation multicast message or it does not hear a periodic advertisement from the bus after a fixed period of time.

When a user clicks on the bus name, the CM tool shows the associated slots. When the user clicks on a slot once, the slot properties are shown in the text window below the slot list. If the user double clicks on a tool, this invokes the tool associated with the slot. The tool information is obtained by connecting to the bus router process and querying it for information about bus slots and other properties. The reconfiguration tools used in POLYLITH [12] use a similar protocol with a centralized bus process.

One application we found this particularly useful in was a spreadsheet connected to a bus slot that relied on a remote database and an external stress analysis program to compute a set of values. The remote database was configured as a required slot while the stress analysis program was configured as an invoked

[2]Van Jacobson of Lawrence-Berkeley Labs calls them "virtual conferences" because they have no origin and are completely decentralized.

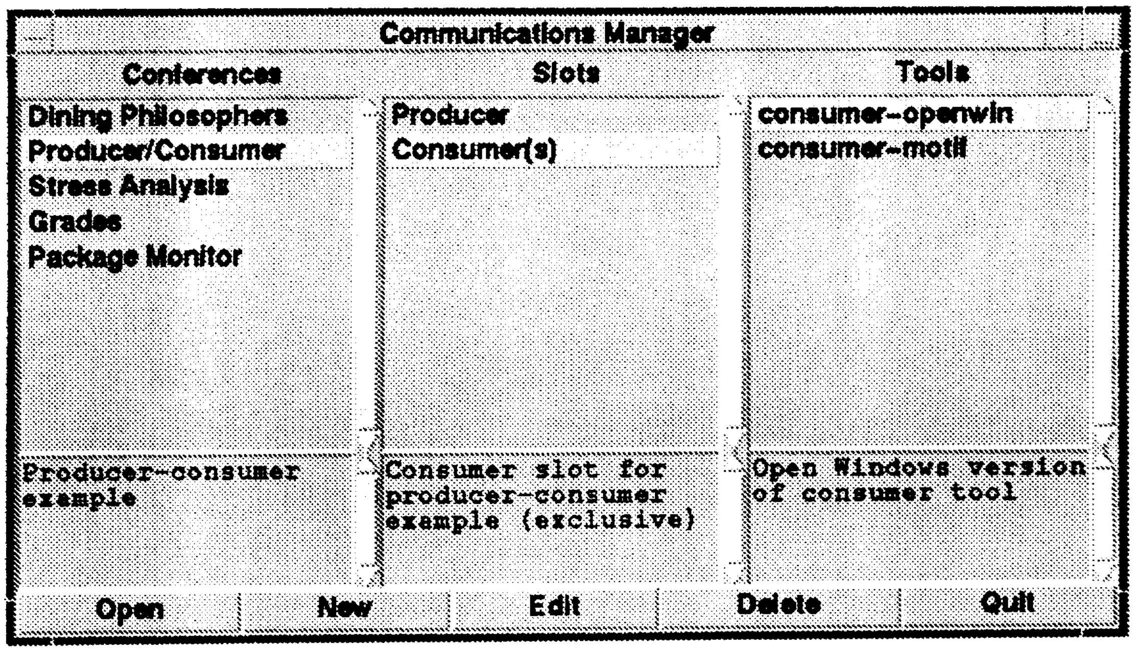


Figure 6: The Communications Manager (CM) application

slot. A delay in registering the database in its slot meant that the spreadsheet queued its messages until they could be processed by the database. Messages sent to the invoked slot could be processed almost immediately. The nice feature of the experiment was that anyone could start up the database from any host on our network and it soon found the bus, enrolled, and began accepting messages without the need for any other process to know its actual location.

A major problem with the centralized bus scheme occurs when slots are unoccupied, but it is desirable to queue messages. The centralized bus must have enough buffer space to queue messages across all interconnections. In a multicast bus, however, each bus occupant is responsible for subsequent retransmissions of queued messages in cases where insufficient positive acknowledgments have been received.

Another major problem with the centralized bus is the difficulty of implementing an interrupt scheme when a message is queued for delivery. Polling by the bus occupant works but it is wasteful. In order to interrupt a bus occupant, the bus process must be co-resident on the same processor or rely on elaborate mechanisms to notify a bus slot of a message. Multicasting, however, requires none of this complexity. Queued messages are continually multicast until their time-to-live expires. A bus occupant can receive a message at any time during the retransmission period. Furthermore, the message can be "requeued" by not responding with an **ACK_DATA**.

Even with **ACK_DATA** messages from recipients, our approach reduces the number of messages needed to send data between bus participants. This assumes that all bus participants know the topology and do not have to query the bus with **REQ_ROUTE** messages. In the case of the producer-consumer, for instance, the producer must receive three **ACK_DATA** messages and one **COUNT_TTL** message in response to one **SEND_DATA** message. Thus, in the MULTIBUS each **SEND_DATA** message requires a minimum of

$$N = 2 + R \qquad (1)$$

messages where R is the number of recipients because a single multicast message is sent and elicits R positive acknowledgments plus the router COUNT_TTL message. Furthermore, there is no overhead of a bus process involved. All R responses are low-level messages that can be sent in parallel. In this case, we assume perfect transmission, but in the worst case, the producer retransmits the message until three acknowledgment are received or a time-out expires. Finally, the MULTIBUS uses the VMTP protocol that has a significantly reduced message overhead than TCP/IP.

In a centralized bus, each data send operation requires a minimum of

$$N = 2 + 2 \times R \tag{2}$$

messages for R recipients. First, the producer must send the message to the bus process. Then, each recipient must be delivered and acknowledge the delivered message. The bus overhead of copying and queuing the message must also be factored into the performance in this case. The $2 \times R$ messages are all handled by the bus process that quickly becomes the bottleneck of the system.

The calculations above are crude and optimistic, but we continue to collect empirical evidence regarding rates of problems including retransmissions, lost messages, and initialization of the client routing tables in the MULTIBUS. This latter cost can be factored out for buses with static topologies since in the limit it will become negligible. In dynamic buses, reconfigurations may be frequent, but the MULTIBUS still has the advantage. A centralized bus or client-server scheme would require complex notifications and negotiations between all parties. In the MULTIBUS, changes to the logical topology are multicast. This allows clients to receive updates asynchronously and maintain up-to-date routing tables.

4 Conclusions

We have described an implementation of a software bus called MULTIBUS using IP multicasting. Our approach increases the performance, reconfigurability, and fault tolerance of a distributed application by relying on IP multicasting to exchange messages between logical hosts on a network. This approach represents an improvement over existing point-to-point methods including remote procedure call. We also describe some motivation for creating a globally visible set of bus conferences without relying on centralized resource services.

Our future plans call for experimentation in a multicast bus without a router process for increased fault tolerance. The interconnection graph will be replicated across a quorum of bus occupants to decrease the probability of failure in the case of partitioning and group failures. In addition, we are implementing authorization mechanisms to control what users are able to see and enroll applications in bus conferences to give further structure and control to interactions between bus participants.

References

[1] Purtilo, J., Polylith: An Environment to Support Management of Tool Interfaces, ACM SIGPLAN Symposium on Language Issues in Programming Environments, Seattle, WA, June 25-28 1985, pp. 12-18.

[2] White, J. E., A High-Level Framework for Network-Based Resource Sharing, AFIPS National Computer Conference, 1976, pp. 561-570.

[3] Nelson, B., Remote Procedure Call, Carnegie Mellon University Department of Computer Science, May 1981.

[4] DeRemer, F. and H. Kron, Programming-in-the-large versus Programming-in-the-small, IEEE Transactions on Software Engineering, June 1976, Volume 2, Number 6, pp. 80-86.

[5] Cheriton, D., Computer Science Department, VMTP: Versatile Message Transaction Protocol, January 1988, Stanford University.

[6] Deering, S. E., Multicasting routing in internetworks and extended LANs, ACM SIGCOMM '88 Symposium, August 1988.

[7] Notkin, D. and A. Black and E. Lazowska and H. Levy and J. Sanislo and J. Zahorjan, Interconnecting Heterogeneous Computer Systems, Communications of the ACM, March 1988, Volume 31, Number 3, pp. 258-273.

[8] Sun Microsystems Computer Corp., Remote Procedure Call Protocol Specification, January 1985, Mountain View, CA.

[9] Magee, J. and J. Kramer, Constructing Distributed Systems in Conic, IEEE Transactions on Software Engineering, June 1989, Volume 15, Number 6, pp. 663-675.

[10] Geist, A., The Portable Virtual Machine (PVM) Environment, University of Tennessee Computer Science Department, April 1990, TR-1677.

[11] Postel, J., Internetwork protocol approaches, IEEE Transactions on Communications, April 1980.

[12] Hofmeister, C. R., Dynamic Reconfiguration in Software Buses, University of Maryland Computer Science Department, August 1993.