

Universal Internet Conference Information System*

H. Abdel-Wahab, I. Stoica and F. Sultan

Department of Computer Science

Old Dominion University

Norfolk, Virginia, 23529

Summary

The Internet Conferencing Information System (ICIS) described in this paper is developed to provide timely information about ongoing computer supported conferences throughout the Internet. ICIS is composed of three major components: Information Server (\mathcal{S}), Conference Announcer (\mathcal{A}) and Conference Querier (\mathcal{Q}). \mathcal{S} stores conference information, \mathcal{A} sends conference information to \mathcal{S} , while \mathcal{Q} retrieves the current conference information from \mathcal{S} and makes it available to the user application. The system is designed to be reliable, efficient and flexible. Reliability is achieved by running multiple well known \mathcal{S} s throughout the Internet. Efficiency is achieved by connecting the servers via a minimum cost spanning tree of TCP connections. Flexibility is achieved by allowing each \mathcal{A} or \mathcal{Q} to communicate with any server using either UDP or TCP protocols. In addition, ICIS protocols are universal in the sense that they are not geared towards any specific conferencing system.

Key Words: Computer Conferencing, Computer Supported Collaborative Work, Internet Protocols, Client/Server Model, Distributed Systems.

*This work is supported by the National Science Foundation Grant # NCR-9313857

1 Introduction

Computer-based conferencing (see, for examples, references 1,2,3,6,9,12,13) is growing due to the widespread use of high performance workstations, the connectivity achieved by the Internet and the interoperability brought by the wide adoption of UNIX and the X Window system.

In order to join an active conference, a user must know the Internet address and the protocol port number where the conference process accepts new connections. The problem is that it is neither convenient nor possible to contact all potential participants in the whole Internet who might be interested in joining a conference and to provide them with the required connection information. It would be more convenient and efficient if such users obtain the needed information of all ongoing conferences by contacting one of a well known set of conference information servers. To achieve this goal, the Internet Conferencing Information System (ICIS) described in this paper is developed to provide real-time information about ongoing computer supported conferences throughout the Internet. ICIS architecture is based on three major components: Information Server (\mathcal{S}), Conference Announcer (\mathcal{A}) and Conference Querier (\mathcal{Q}). \mathcal{S} s store conference information generated by \mathcal{A} s, while \mathcal{Q} s retrieve the information which helps the user to join any of these ongoing conferences.

To achieve both reliability and availability we run multiple well known \mathcal{S} s throughout the Internet. An \mathcal{A} contacts any close by \mathcal{S} and in turn that \mathcal{S} multicasts the received information to all other \mathcal{S} s via a minimum-cost spanning tree connecting all \mathcal{S} s. Because the operating system usually imposes a limit on the number of TCP connections to any given process [7], we allow the clients (\mathcal{A} s and \mathcal{Q} s) to choose between UDP and TCP in communicating with \mathcal{S} s. In addition, ICIS services are universally available since its protocols are not geared towards or tailored for any specific conferencing system. By using ICIS, it will be easy for any user to start a new conference focused on a specific topic and give users across the Internet the opportunity to join that conference.

Section 2 is an overview of ICIS components and architecture. Section 3 deals with ICIS protocol and communication issues. Section 4, 5 and 6 describe the \mathcal{A} , \mathcal{Q} and \mathcal{S} components, respectively. Finally, in section 7 we present our conclusions.

2 System Overview and Architecture

The ICIS system is based on a client-server model, where the server is replicated at multiple sites. Replication provides a means for increasing both reliability and availability of the ICIS services. Reliability of a service means that the service must be fault-tolerant up to a certain limit, defined as the *resilience degree* of the system. The resilience degree is the maximum number of faulty sites for which the functionality of the system is not affected. Availability of a service is defined as the possibility of transparently accessing the service facilities at all times, without being concerned with where and who is implementing the service. If one of the servers crashes or for some reason it is not running, other servers can provide the same information about all ongoing conferences in the Internet. Figure 1 shows an example that depicts the overall view of the three ICIS components.

Each conference announcer \mathcal{A} or querier \mathcal{Q} selects a server \mathcal{S} to be contacted for subsequent reporting or querying. The selection process is not a random one: it tries to minimize an objective function based on the communication delay and the estimated loads of \mathcal{S} s. In this way, over a period of time, all servers are evenly loaded and, at the same time, responsiveness to \mathcal{A} s and \mathcal{Q} s is improved.

Every conference has a unique conference identifier across the Internet, consisting of host Internet address and conference port number. For each conference, \mathcal{S} s maintain associated information such as start time, conference coordinator, subject, participants and any other information that the conference participants may choose to publicize.

Either a connection-oriented (TCP) or a connectionless (UDP) protocol can be used as the support for the communication among the clients (\mathcal{A} s and \mathcal{Q} s) and the servers (\mathcal{S} s). In ICIS we use a form of UDP called *reliable* UDP where every message is guaranteed to be delivered. Reliable UDP uses acknowledgments, time-outs, retransmissions and sequence numbers to achieve the guaranteed delivery requirements. See [14] for details on how to provide a reliable UDP service based on the standard unreliable UDP protocol.

The servers are connected together by a minimum-cost spanning tree (MST) of TCP connections. In constructing the MST we consider the communication network modeled as a connected graph, where the servers represent the nodes in the graph. To each edge we associate as weight the transmission delay between end-nodes. Our algorithm for building

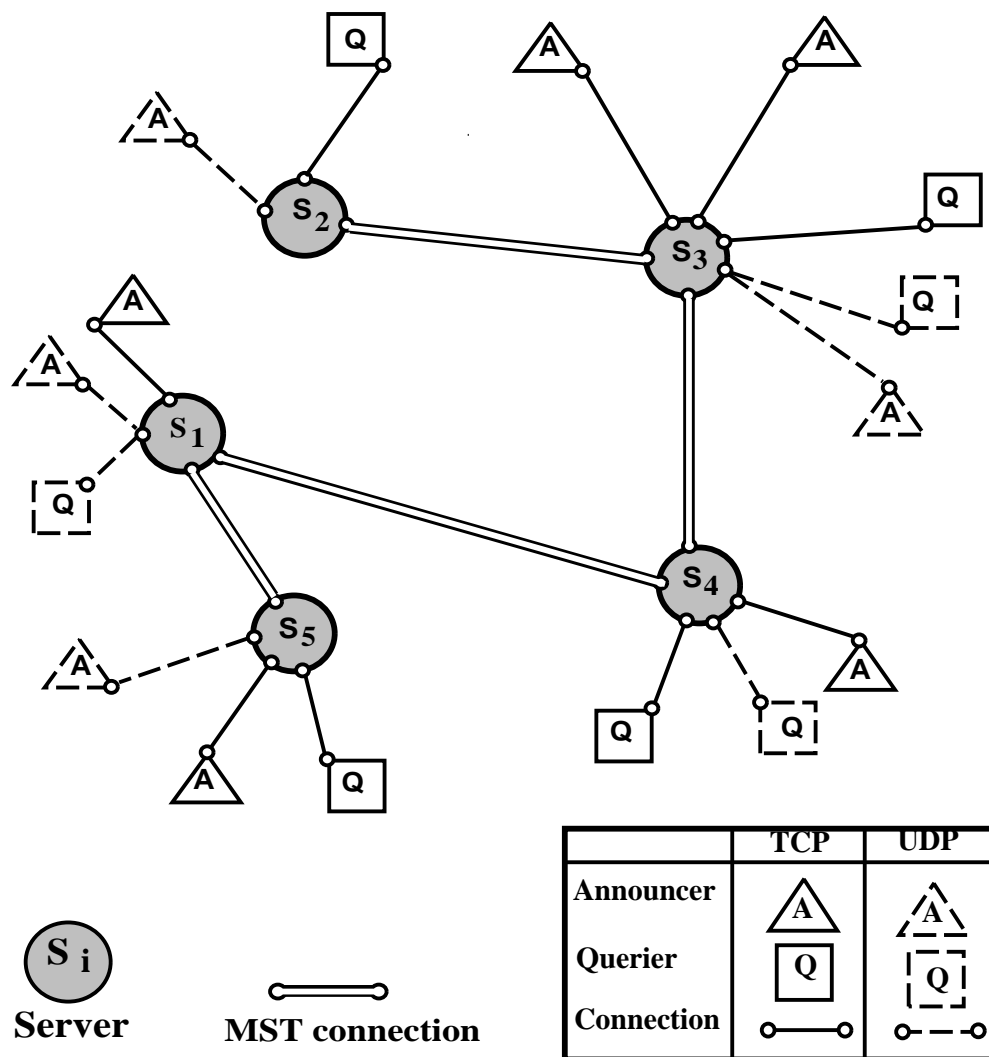


Figure 1: ICIS components: S_s , \mathcal{A} s and \mathcal{Q} s

the MST in this graph, described in [4], is totally distributed and is based on the idea of maintaining a collection of disjoint trees (forest) during the execution. Initially, every tree consists of only one node (server). As the algorithm proceeds each tree independently searches for the closest tree in the forest and merges with it. The algorithm terminates when all the servers are connected in a single tree which is the final MST.

In Figure 1 there are five servers connected with a spanning tree and a total of sixteen clients distributed over the five servers. For instance, server S_3 has five clients (three \mathcal{A} s and two \mathcal{Q} s). Three clients are connected to S_3 with TCP connections while the remaining two clients are using UDP.

The main problem arising when maintaining replicated information like the one kept by

Message Type	Fields	Description
CINFO	cid, cinfo	send conference info to \mathcal{S} s
TERMINATION	cid	notify \mathcal{S} s of \mathcal{A} 's normal termination
REQUEST_ALL_CINFO		request conference info list from an \mathcal{S}
ALL_CINFO	cinfo_list, cid_list	receive conference info list from an \mathcal{S}
UPDATE_NOTICE		notify conference info change to \mathcal{Q}
ARE_YOU_ALIVE	process_id	aliveness check

Figure 2: **Protocol Message Types for ICIS**

\mathcal{S} s is the problem of consistent updates. In a real database this would involve maintaining a consistent view of the database for all \mathcal{Q} s, such that individual transactions on different replica will give the same result. In the case of ICIS the corresponding problem would be to maintain the same information by all \mathcal{S} s at all times, such that any \mathcal{Q} will obtain the same conference information, no matter which of the \mathcal{S} s provides it. However, this very restrictive constraint could be very expensive to implement. Implementing complex commitment protocols would be ultimately required. In ICIS the strong consistency requirement of the replicated database is relaxed: the main objective of the system is to maintain “asymptotically” consistent replica of the same information. This means that at a give instant of time T the information available at all servers may not be identical. However, if there are no updating activities for a period of time $\Delta T > 0$, then all servers will have the same information. The time interval ΔT depends on the network delays and the processing time needed to propagate the information between the servers along the minimum spanning tree connections. In our application, it is desirable but not necessary that all \mathcal{Q} s have the same information. This is different from most other distributed database applications (e.g., banking and stockmarket applications), where it is absolutely necessary for all clients to see the same data.

3 ICIS Protocol and Communication Issues

The service is available to clients (\mathcal{Q} s and \mathcal{A} s) through both TCP and UDP. Problems related to the reliability of the client-server communication in the UDP case is solved

by employing a *reliable* UDP implementation. Since in both cases all messages sent are acknowledged, one party is guaranteed to be informed about the other's aliveness. However, in situations of severe failure, when the TCP cannot provide timely aliveness information, other mechanisms must be provided to ensure that a process has always up-to-date information about the aliveness of its peers. This involves periodically checking the status of the communication links (and implicitly of the peer) by means of special messages for detecting aliveness.

We note that in the case of an information system like ICIS there are no hard bounds for the delay with which users have to be provided with timely information. However, responsiveness of the system is still important: information from \mathcal{A} s has to propagate with reasonable delay and presence of stale information at \mathcal{Q} s should be avoided. For example, if a server crashes, its attached clients need to become aware of this fact and take appropriate actions (e.g., switch to another server). From the point of view of a given client's expectations, a server crash is no different from a transient network failure due to which the server becomes unreachable. Thus, ICIS does not need to address the general problem of fault detection in a distributed system, but rather tries to react rapidly to failures of either type by employing a simple mechanism based on periodic checks of aliveness of one communicating party by the other. In case of the failure of the underlying transport protocol in delivering either these messages or any other ICIS protocol message to a peer process, it is assumed that the corresponding party has suffered a severe failure and specific actions have to be taken.

The TCP protocol software provides its own mechanisms for detecting aliveness by one of the two parties involved in communication using the *retransmission* and *keep-alive* timers [15]. However, these TCP mechanisms do not meet the real-time responsiveness required by some applications like ICIS. For example, if the machine on which one end point of a TCP connection crashes, using the keep-alive timer would imply an unacceptable delay (about 2 hours [15]) between the moment of the crash and the moment of detecting it at the other endpoint. Periodically pinging the other peer by special ARE_YOU_ALIVE messages would seem to be a reasonable solution. However, these messages have to be explicitly acknowledged at the application level by the other party; relying on the TCP acknowledgment mechanism and retransmission timers would fail with the same result

(long delay in detecting the crash). This is primarily due to slowing down retransmissions by TCP in response to unacknowledged segments as part of its internal congestion avoidance mechanism.

Unlike TCP, a reliable UDP protocol based only on simple acknowledgment and retransmission mechanisms implemented at the user process level cannot detect and inform one of the communicating parties that its peer has terminated. The problem is only partially solved by periodically pinging the server to check if it is alive or not. This is because multiple retries have to be employed to account for temporary failures of the communication link, when datagrams or their acknowledgments are lost. However, if the server goes down and then is quickly restarted, there would be no way for the UDP clients to discriminate between the two incarnations of the server and to take appropriate actions. To overcome this situation, a simple solution would be to have the clients be aware of the specific identity of the server (by means of a unique identifier, e.g., its process id) and tag its ping messages with this identifier. When a server detects that a message is not addressed to it, it can just warn its client, and the client will take appropriate actions.

Finally, we describe the strategy used to evenly distribute the load among servers while maintaining low communication costs. A mechanism based on a linear weighted function of server load and round trip time is employed by \mathcal{A} s and \mathcal{Q} s for selecting a “good” \mathcal{S} to communicate with. Messages to allow clients to query the load of potential servers to be contacted and compute this function are provided in the ICIS protocol.

Intuitively, a client should try to select a “nearby” and “lightly loaded” server \mathcal{S} . Quantitatively, these combined requirements are reflected by the optimal (minimum) value of a linear weighted function of the round-trip time (RTT) and server load (L) parameters:

$$f = \beta L + (1 - \beta)RTT \quad (1)$$

where the weighting factor β is a positive value less than one reflecting the importance assigned to the communication and to the load as factors which impact the responsiveness of the system. Minimizing this function over the set of the available \mathcal{S} s will yield a best choice in terms of both of these factors.

For a given server, the number of TCP connections to its clients may be limited. Thus, besides selecting a server, a client will have to negotiate the type of transport protocol that the server can support. If a certain limit of simultaneously open TCP connections is reached by a server then the service will fall back to being provided through UDP (by means of a simple negotiation protocol). This is because the overhead is much lower to establish a single UDP port open for the entire life of \mathcal{S} and being able to respond to multiple destination addresses than having to establish and maintain individual TCP connections for a possibly large number of clients.

Figure 2 shows all the ICIS protocol messages. The algorithms executed by \mathcal{A} , \mathcal{Q} and \mathcal{S} are described in the following three sections.

4 Conference Announcer Procedure

The basic algorithm performed by every conference announcer (\mathcal{A}) is shown in Figure 3. Each \mathcal{A} maintains a list of the closest information servers (\mathcal{S} s) in ICIS. We assume that at any time at least one \mathcal{S} from this list is operational. When \mathcal{A} starts, it first selects an \mathcal{S} from its list and connects to it. If there is more than one \mathcal{S} available, then a special weighted function based on the round trip delay and the server load is used to select the “best” \mathcal{S} as we have described in Section 3. \mathcal{A} may communicate with the selected \mathcal{S} either through a TCP connection or through reliable UDP. Establishing a TCP connection has a much higher overhead than using UDP and the number of TCP connections supported by \mathcal{S} is limited. When this limit is reached \mathcal{S} will offer its service only through UDP.

Next, \mathcal{A} sends a CINFO message containing the conference identifier (*cid*) and other conference related information (*cinfo*) to the selected \mathcal{S} . When CINFO is sent for the first time, *cinfo* field contains “complete” conference related information. Otherwise, it contains only “differential” information such as the new, changed and deleted information since the last CINFO message was sent. The traffic on the communication network is minimized by sending to \mathcal{S} only incremental changes. Obviously, the amount of network traffic saved depends on the specific conferencing system represented by \mathcal{A} .

Whenever the conference information is changed, a new CINFO message is sent to \mathcal{S} , which, in turn, updates the conference information in its local data structures. Upon

protocol message types

CINFO[*cid*, *cinfo*, ...], TERMINATION[*cid*, ...]

variables

current_cinfo, *old_cinfo*, *cid*

1. **select** \mathcal{S} 2. CINFO.*cid* := *cid*

CINFO.*cinfo* := *current_cinfo*

send CINFO to \mathcal{S} // *send complete conference information*

old_cinfo := *current_cinfo*

3. **loop-forever**3.1. **on** *current_cinfo* change

CINFO.*cinfo* := **diff**(*current_cinfo*, *old_cinfo*) // *send only differential information*

send CINFO to \mathcal{S}

old_cinfo := *current_cinfo*

3.2. **on** conference termination

send TERMINATION to \mathcal{S}

3.3. **on** time-out

check aliveness of \mathcal{S}

3.4. **on** failure to send message to \mathcal{S} // *the server or the communication link has failed*

execute step 1 and step 2

Figure 3: \mathcal{A} - Announcer algorithm

receiving this message for the first time, \mathcal{S} creates a new record for \mathcal{A} and inserts the received conference information in the conference information list (see Figure 6, $\mathcal{S}_{\mathcal{M}}$: case a). Otherwise it simply updates the corresponding entry in the conference information list. Every \mathcal{S} maintains a global version number (*info-list-version*) that is incremented whenever a new conference is created or an old one is updated or terminated. The main purpose of maintaining the version number is to allow an efficient exchange of information with the conference queriers as will be explained in more detail in Section 5.

When a conference terminates, \mathcal{A} sends a TERMINATION message to \mathcal{S} . Upon receiving this message, \mathcal{S} deletes \mathcal{A} 's related conference information from its list.

Periodically, \mathcal{A} sends an ARE_YOU_ALIVE message to \mathcal{S} in order to check whether \mathcal{S}

is still alive. If \mathcal{A} fails in sending this message (i.e. it is not acknowledged by the server) or any other message to \mathcal{S} , then \mathcal{A} assumes that the corresponding \mathcal{S} is down and the procedure to select a new \mathcal{S} is started.

5 Conference Querier Procedure

The basic algorithm performed by a conference querier (\mathcal{Q}) is depicted in Figure 4. Each \mathcal{Q} manages the following main data structures:

- *current_cinfo_list* contains the local copy of the conference information list.
- *update_flag* is a flag specifying whether or not \mathcal{Q} 's conference information is up-to-date. Its value is *FALSE* if and only if \mathcal{Q} has an old copy of the conference information list.

Like an \mathcal{A} , each \mathcal{Q} starts by selecting an operational \mathcal{S} from its list. Again, we assume that there is at least one operational \mathcal{S} in every \mathcal{Q} list and therefore an \mathcal{S} can always be selected. When an \mathcal{S} is selected, a similar weighted function based on the server load and the round trip delay, as in \mathcal{A} 's case, is used. Also, both types of transport protocols (TCP and reliable UDP) are available for establishing the connection with \mathcal{S} .

After selecting and contacting an \mathcal{S} , \mathcal{Q} sends a REQUEST_ALL_CINFO message in order to obtain the information related to all ongoing conferences in ICIS. Upon receiving this message for the first time, \mathcal{S} creates a record for the new \mathcal{Q} and associates a version number, initialized to 0, to it (see Figure 6, $\mathcal{S}_{\mathcal{M}}$: case b). The \mathcal{Q} 's version number represents the version of the last conference information copy that was sent to \mathcal{Q} . If the \mathcal{Q} 's version number is equal to 0, this means that \mathcal{Q} has no information about any ongoing conferences. Further, \mathcal{S} builds an ALL_CINFO message containing two main fields called *cinfo_list* and *cid_list* and sends it back to \mathcal{Q} . The *cinfo_list* contains the information about all ongoing conferences that have been created or modified since the last ALL_CINFO message was received by \mathcal{Q} , while *cid_list* contains the list of all unmodified conferences. Next, the version number in \mathcal{Q} 's record is updated to the current version of the conference information maintained by \mathcal{S} (i.e. \mathcal{Q} has now an up-to-date version of the conference information list). Also, associated with every \mathcal{Q} , \mathcal{S} maintains a flag, called *update_flag*,

protocol message types

REQUEST_ALL_CINFO[...], ALL_CINFO[cinfo_list, cid_list, ...], UPDATE_NOTICE[...]

variables

update_flag, old_cinfo_list, current_cinfo_list, r

1. **select** \mathcal{S}

2. **send** REQUEST_ALL_CINFO to \mathcal{S}

3. **loop-forever**

3.1. **on** receiving \mathcal{S} protocol message

a) **case** ALL_CINFO:

old_cinfo_list := current_cinfo_list // *save the current list*

current_cinfo_list := ALL_CINFO.cinfo_list

for every ((r in old_cinfo_list) and (r.cid in ALL_CINFO.cid_list))

insert r in current_cinfo_list // *insert the unmodified conference in list*

update_flag := *TRUE*

deliver current_cinfo_list to user

b) **case** UPDATE_NOTICE:

update_flag := *FALSE*

notify user

3.2. **on** time-out // *check if \mathcal{S} is alive*

if update_flag = *TRUE*

check aliveness of \mathcal{S}

3.3. **on** user request

if update_flag = *FALSE*

send REQUEST_ALL_CINFO to \mathcal{S}

3.4. **on** failure to send message to \mathcal{S}

execute step 1 and step 2

Figure 4: \mathcal{Q} - Querier algorithm

that specifies whether or not Q has the last version of the conference information. Once the ALL_CINFO message is sent, the *update_flag* flag of the corresponding Q is set to *TRUE* (i.e., the Q information is up-to-date).

Upon receiving an ALL_CINFO message, Q builds the new conference information list by adding the information for the newly created conferences and updating the information of the conferences that have been modified. In this way all the conferences that are in the old list, but are not in the *cid_list* are removed (i.e. if a conference is in the old list but not in *cid_list*, this means that the conference has been terminated).

When the conference information at S is changed as a result of creating, updating or deleting a conference, S sends to Q an UPDATE_NOTICE message. Upon receiving this message Q sets the *update_flag* flag to *FALSE* and notifies the user that an information change has occurred. The user may or may not be willing to receive the new updated information from S . On user's request to update the conference information, Q sends a REQUEST_ALL_CINFO message to S .

If Q has not received any message from S for a long time (i.e. when a time-out expires) and the value of the *update_flag* is *TRUE* (indicating that the user might be interested in obtaining the updated information), then Q checks if the selected S is still alive by sending an ARE_YOU_ALIVE message to S . If in the meantime the corresponding S went down, then Q starts the procedure to select a new S and to obtain a new copy of the conference information list.

6 Information Server Procedure

The main purpose of an information server S is to maintain an accurate copy of all the conferences that run across the Internet. All S s in ICIS are connected through TCP connections in a spanning tree topology. Since all S s must maintain a consistent conference information list, every time an S updates the information about a conference it multicasts this change to all the other S s along the spanning tree.

Figures 5 and 6 describe the algorithm performed by an information server (S). Every S manages the following data structures and variables:

- *info_list* contains the information about all ongoing conferences in the Internet.

protocol message types

include all messages from Figure 2

variables

info_list, info_list_version, Q_List, A_List, MST_adj_list, r

1. loop-forever

1.1. **on** receiving $\mathcal{A}/\mathcal{Q}/\mathcal{S}$ protocol message

See Figure 6 for message handling

1.2. **on** time-out

for each A **in** A_List

check aliveness of A

for each Q **in** Q_List

check aliveness of Q

1.3. **on** failure to send message to $\mathcal{A}/\mathcal{Q}/\mathcal{S}$

case \mathcal{A} :

delete \mathcal{A} **from** A_List

TERMINATION.cid = \mathcal{A} 's cid

execute step 3 // *Delete \mathcal{A} 's information*

case \mathcal{Q} :

delete \mathcal{Q} **from** Q_List

case \mathcal{S} :

reconfigure the network spanning tree

2. for each Q **in** Q_List

if Q.update_flag = *TRUE*

Q.update_flag := *FALSE* // *Q does not have the latest information*

send UPDATE_NOTICE to Q

3. delete info_list[TERMINATION.cid] **from** info_list

increment info_list_version

for each S **in** MST_adj_list **except** sender

send TERMINATION to S

execute step 2 // *notify Qs*

Figure 5: \mathcal{S} - Server algorithm

```

a) case CINFO:
  if (sender is  $\mathcal{A}$  and (sender not in A_List))
    insert sender in A_List
  if CINFO.cid not in info_list
    insert (CINFO.cid, CINFO.cinfo) in info_list
  else
    update info_list using (CINFO.cid, CINFO.cinfo)
  increment info_list_version
  r.version:= info_list_version // r is A's record in info_list
  for each  $\mathcal{S}$  in MST_adj_list except sender
    send CINFO to  $\mathcal{S}$  // forwards the conference info along the spanning tree
  execute step 2 // notify  $Q$ s
b) case REQUEST_ALL_CINFO:
  if sender not in Q_List
    insert sender in Q_List
    Q_List[sender].version:= 0 // Q has no information about any ongoing conf.
  for each record r in info_list
    if r.version > Q_List[sender].version
      insert r in ALL_CINFO.cinfo_list
    else
      insert r.cid in ALL_CINFO.cid_list
  send ALL_CINFO to sender
  Q_List[sender].update_flag:= TRUE
  Q_List[sender].version:= info_list_version
c) case TERMINATION:
  if sender is  $\mathcal{A}$ 
    delete sender from A_List
  execute step 3 // Delete A's information

```

Figure 6: $\mathcal{S}_{\mathcal{M}}$ - Handling Protocol Messages in \mathcal{S}

Whenever a new conference is created or an old one is updated or terminated, a variable called *info_list_version* is incremented. Each record *a* in the *info_list* contains a field, called *a.version*. If the conference information associated with a record is changed, the *a.version* field is set to the *info_list_version*.

- *A_List* contains the list of all \mathcal{A} s connected to \mathcal{S} .
- *Q_List* contains the list of all \mathcal{Q} s connected to \mathcal{S} . Each record *q* contains a *q.version* and a *q.update_flag* field. The *q.version* field contains the version number of the last conference information list that has been sent to the corresponding \mathcal{Q} . The *q.update_flag* specifies whether or not the corresponding \mathcal{Q} has the latest conference information list. Thus, this flag is set to *FALSE* whenever the *info_list_version* is greater than the *q.version* field indicating that \mathcal{Q} has an old copy of the conference information list, otherwise \mathcal{Q} has an up-to-date copy of the conference information list.
- *MST_adj_list* contains a list of all neighbors of \mathcal{S} s in the minimum spanning tree. Whenever the information related to a specific conference is changed as a result of receiving a CINFO message (a new conference is created or an old one is modified) or a TERMINATION message (a conference is terminated), then an \mathcal{S} forwards the message to all neighbors (i.e. all \mathcal{S} s in the *MST_adj_list* excluding the sender if it is an \mathcal{S}). Notice that if the message is received from an \mathcal{A} then it is forwarded to all the neighbors, since *MST_adj_list* contains only \mathcal{S} s.

Upon receiving a CINFO message, \mathcal{S} checks if the sender is an \mathcal{A} . If this is the case and the sender is not in *A_List*, then it is added to the list. Next, \mathcal{S} checks if the conference has been previously registered. If not, a new record containing all conference related information is added to the *info_list*. Otherwise (if the conference has already been registered), the conference related information is changed in the *info_list* (recall that the *cinfo* field from CINFO message contains, in this case, only incremental changes to the conference information and not the complete information). In both cases \mathcal{S} increments the version number (*info_list_version*) of the *info_list*. Since all \mathcal{S} s have to know about this change, \mathcal{S} forwards the message to all its neighbors in the spanning tree (the \mathcal{S} s in the

MST_adj_list). Finally, \mathcal{S} tests the *update_flag* flag in every \mathcal{Q} 's record and if its value is *TRUE* (indicating that \mathcal{Q} is genuinely interested in having the latest information) it sets it to *FALSE* and notifies the corresponding \mathcal{Q} by sending an UPDATE_NOTICE message.

When a REQUEST_ALL_CINFO message is received from a \mathcal{Q} , then an ALL_CINFO message is built. As shown before, this message contains two main fields: *cinfo_list* and *cid_list*. The *cinfo_list* field contains the information related to all the conferences that have been created or modified since the last ALL_CINFO message was sent to \mathcal{Q} (i.e. all records in the *info_list* with the version greater than the *version* field in \mathcal{Q} 's record), while the *cid_list* contains the identifier of all unchanged ongoing conference (this list is used by \mathcal{Q} to determine all the conferences that have been terminated since the last time it has received an ALL_CINFO message). When ALL_CINFO message is sent to \mathcal{Q} , the *update_flag* flag of \mathcal{Q} 's record is set to *TRUE* and the *version* field is updated to the current conference information list version (*info_list_version*).

Upon receiving a TERMINATION message, \mathcal{S} deletes the corresponding \mathcal{A} 's entry from its *A_List*. In addition, it deletes the associated conference information from *info_list*, increments the *info_list_version* and forwards the TERMINATION message to all its \mathcal{S} neighbors. Then it notifies the interested queriers (those having the *update_flag* flag set to *TRUE*) that the *info_list* has changed and sets the corresponding *update_flag* flag of those \mathcal{Q} to *FALSE*.

To ensure that all \mathcal{A} s and \mathcal{Q} s are still alive, \mathcal{S} periodically sends an ARE_YOU_ALIVE to every registered \mathcal{A} and \mathcal{Q} . On failure to contact a \mathcal{Q} , \mathcal{S} simply deletes it from the list. The corresponding action for \mathcal{A} is slightly more complicated. First, \mathcal{A} 's record and the corresponding conference information are deleted from its *A_List* and *info_list* respectively. Second, *info_list_version* is incremented and a TERMINATION message is generated and sent to all its \mathcal{S} neighbors. In this way, all the other \mathcal{S} s in the system will remove the terminated conference from their *info_lists* and therefore the consistency is ensured. Finally, \mathcal{S} informs the interested \mathcal{Q} s in its *Q_List* of the information change.

7 Concluding Remarks

Due to the recent advances in the field of Computer Supported Cooperative Work [5,11] computer conferencing is now a growing application in the Internet. The ICIS system presented in this paper provides a real-time information service for the Internet users who are interested in finding information about any of these collaborative conferences. Users may then use this information to join any conference of interest to them (e.g., based on the conference subject and participants). In order to increase the efficiency, reliability and availability of the system, multiple servers are executed concurrently at different locations in the Internet and are connected together using a minimum cost spanning tree. We have implemented and experimented with the protocols presented in the paper in the context of the XTV conferencing system [2,3,6]. In our implementation we have provided an X/Motif window interface for ICIS to query and join any ongoing XTV conference. The source code of our implementation is publicly available via anonymous ftp from Old Dominion University. We strongly believe that the full scale adoption and implementation of ICIS protocols will increase the level of cooperation and the frequency of real time interaction among the Internet community of users.

During the past years conferencing over Internet benefited from the deployment of the MBone and use of IP wide-area multicast[16]. While offering a general solution, a system based only on wide area multicast to a densely populated wide area group is likely to be affected by problems inherent with current IP multicast protocols, the biggest being that of scalability. Efforts are currently being made to develop support for scalable multicast [8,16]. By employing a custom communication structure in which the server replica are interconnected, the ICIS architecture is independent of the underlying multicast IP support. Independence of the multicast transport mechanism addresses two other problems. The first is that of availability: while a site needs to be a node in the MBone to take advantage of its multicast IP capabilities, any host in the Internet can run one of the ICIS components (\mathcal{S} , \mathcal{Q} and \mathcal{A}) without being required to set up and configure a router. The second issue is that of reliability: the inter-server transport in ICIS is reliable, while the problem of scalable wide-area reliable IP multicast has only recently begun to receive reasonable solutions [10]. However, we note that the communication

structure that ICIS is currently using for interconnecting servers can be easily replaced, given the support of scalable and reliable wide area IP based multicast. A wide area group comprising only ICIS servers would be an immediate first step towards integrating IP multicast in ICIS.

Although the ICIS system is specifically designed and implemented to handle conference information, the communication architecture and part of its protocols can be also used in any other application in which dynamic information has to be disseminated in real time to a potentially large number of receivers, spread across a wide area in the Internet.

References

1. H. Abdel-Wahab, S-U. Guan, and J. Nievergelt, J., 'Shared Workspaces for Group Collaboration: An Experiment using Internet and UNIX Interprocess Communications', *IEEE Communications Magazine*, Vol. 26, No. 11, 10-16 (November 1988).
2. H. Abdel-Wahab and M. A. Feit, 'XTV: A Framework for Sharing X Window Clients in Remote Synchronous Collaboration', *Proceedings, IEEE Conference on Communications Software: Communications for Distributed Applications & Systems*, Chapel Hill, North Carolina, 159-167 (April 1991).
3. H. Abdel-Wahab and K. Jeffay, 'Issues, Problems and Solutions in Sharing X Clients on Multiple Displays', *Internetworking Research and Experience*, Vol. 5, No. 1, 1-15, (March 1994).
4. H. Abdel-Wahab, I. Stoica and F. Sultan, "A Distributed Algorithm to Compute the Minimum Spanning Tree in a Communication Network", To appear, *Proceedings of the Second International Conference on Computer Theory and Informatics*, Wrightsville Beach, NC, Sept 1995.
5. R. M. Baeker, *Readings in Groupware and Computer-Supported Cooperative Work, Assisting Human-Human Collaboration*, Morgan Kaufmann Pub. Co., Palo Alto, CA (1993).
6. G. Chung, K. Jeffay and H. Abdel-Wahab, 'Accommodating late-comers in shared window systems', *IEEE Computer*, Vol. 26, No. 1, 72-74, (January 1993).

7. D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP Volume III: Client-Server Programming and Applications: BSD Socket Version*, Prentice-Hall, (1993).
8. S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu and L. Wei, 'An Architecture for Wide-Area Multicast Routing', *Proceedings of the ACM SIGCOMM '94*, August 1994.
9. P. Dewan and R. Choudhary, 'A high-level and flexible framework for implementing multiuser interfaces', *ACM Transaction on Information Systems*, Vol. 10, No. 4, 345-380, (October 1993).
10. S. Floyd, V. Jacobson, S. McCanne, C. Liu, L. Zhang, 'A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing', To appear, *Proceedings of the ACM SIGCOMM '95*, Boston, MA, August 1995.
11. J. Grudin, 'Computer-Supported Cooperative Work: History and Focus', *IEEE Computer*, Vol. 27, No. 5, 19-26, (May 1994).
12. J. C. Lauwers and K. A. Lantz, 'Collaboration Awareness in Support of Collaboration Transparency: Requirements for the Next Generation of Shared Window Systems', *Proceedings, Conference on Human Factors in Computer Systems*, ACM Press, 303-311 (April 1990).
13. J. F. Patterson, R. D. Hill, S. Rohall and W. S. Meeks, 'Rendezvous: An architecture for synchronous multi-user applications', *Proceedings of the Third Conference on Computer-Supported Cooperative Work*, ACM Press, 317-328 (October 1990).
14. W. R. Stevens, *Unix Network Programming*, Prentice-Hall, (1990).
15. W. R. Stevens, *TCP/IP Illustrated Vol 1*, Addison-Wesley, (1994).
16. A. S. Thyagarajan and S. E. Deering, 'Hierarchical Distance-Vector Multicast Routing for the MBone', To appear, *Proceedings of the ACM SIGCOMM '95*, Boston, MA, August 1995.