CHATNET - a Distributed Multi-channel Conferencing Protocol

Gerrit Hiddink
University of Twente
January 1995

ABSTRACT
The aim of this paper is to give insight into the problems of the IRC protocol, and how they are solved by the proposed protocol, called Chatnet. It also explains the structure of Chatnet, how it is divided into functional parts and how these parts interact. Special attention is given to the Multicast layer, and what unsolved problems still exist. Also the error-correcting properties of the protocol are explained. Finally some concluding remarks are given.

About the author
The author is a Computer Science graduate student at the University of Twente in Enschede, Holland. His fields of interest are multicast routing, distributed computing and internetworking. His graduation assignment will have something to do with some of these three, applied to ATM networks. The work presented here, however, was conducted in his spare time.
E-mail address: hiddinkg@pegasus.esprit.ec.org

This paper is to be presented at Gronics '95, 24 february 1995, Groningen, The Netherlands.

1. What is a distributed multi-channel conferencing protocol anyways?

A distributed system is a system that consists of processing elements with communication channels between them. A distributed protocol describes the way the processing elements communicate via these channels. A conferencing protocol is a set of rules that describes the message exchange of a system that is built to let people conference with eachother. A multichannel conferencing protocol is a protocol for a conferencing system in which people can talk in channels (sometimes called groups). Conferencing systems are mostly based on the client-server model. The conferencing system that is mostly used on the Internet, Internet Relay Chat or IRC for short, is also based on this model; its main property is that the servers maintain all data necessary to run the system, and that clients connect to them to request this data. Servers are permanently coupled using communication links, mostly TCP connections. A good way to model a distributed system is the following: servers are nodes in a graph. The edges are the communication links between them. If the graph is connected, then information between all nodes can be exchanged. In some systems, servers use explicit knowledge of the graph in order to route information through the graph. For example, the graph of the IRC protocol is a spanning tree. When carefully chosen, a spanning tree can provide a fast and efficient way to distribute information through the graph.

2. Problems of existing conferencing protocols

The forementioned IRC protocol is described in RFC1459. Currently over 4000 users are on-line at a time, using about 100 servers; conversations are grouped into about 1500 channels. The pro-

tocol, however, does not scale to this amount of users. Its main problems will be treated below.

2.1 Spanning tree

The IRC protocol uses a spanning tree to distribute information. A spanning tree formally is a graph of nodes (servers) and edges (communication links) that spans every server, and does not contain cycles. This means there is only one way to reach a certain destination. Messages cannot be duplicated. As said before, a spanning tree can be a very efficient way to distribute information. There is however one major drawback: whenever a communication link breaks, the tree is split in two parts. Both will continue functioning normally, but their state information (user and channel tables etcetera) will become different. These two versions then should be merged again when the link is mended, which is not a trivial question.

The probability of a link breaking grows linearly with the number of links. This means that the larger such a system becomes, the more instable it will be.

Finding the most efficient spanning tree needs explicit knowledge of the underlying network service, e.g. what network links are fastest. This knowledge is often obtained by average ping times. Then it is decided where the new server must be linked in, and it remains there for the rest of its lifespan. If the network characteristics change however, parts of the tree should be rebuilt to obtain the most efficient spanning tree. The IRC protocol has no provisions to do this automatically, so that maintaining the tree is a lot of work in large systems, and it becomes increasingly more work as the network grows. In other words: it does not scale very well.

The spanning tree also does not adapt to congested servers, as it is static. Heavily loaded servers cause long ping times, so that pings time out and the link to the congested server is disconnected, thus splitting the tree. When both parts are joined again, they start exchanging their state information. This can cause a very high load in a large system, causing ping timeouts etcetera. The possibility that this process repeats over and over again is quite high. The heavy task of maintaining the IRC network has been the cause of fights between the IRC operators in the past. This has in a few cases led to a division in which a group of operators created their own IRC network. Currently two relatively large IRC networks are in operation (4000 and 500 users, respectively called EFnet and Undernet), as well as several small ones. These unfortunate events have caused IRC to be viewed as a "young people's toy" not suitable for professional conferencing, as some fights were guided by link breaks caused by IRC operators themselves. It is hoped that the proposed protocol will become a tool as useful as any other common Internet resource.

2.2 State information

The above mentioned problems would not be as sincere if the state information kept by every server would be minimal. Every server, however, keeps information about every user, every channel, every server, and what user is on what channels with what user modes. It is clear that the amount of state information scales linearly with the number of users: twice as much users need twice as much server memory to store the state information. The inter- server traffic between any two servers also scales linearly, as every change in the characterstics of any user is distributed to all servers (eg. a change in nickname, usermode, or a change in the channels the user is on). Due to this bad scaling, the protocol will cause physical limits to be reached when scaled infinitely. For example, a communication link may be unable to carry the amount of traffic needed to update the global state information.

3. Solving these problems

3.1 link breaks

If we want to avoid the effect of increasing probability that a link breaks, we must supply redundant links. This causes cycles (from graph theory we can learn that a tree becomes cyclic if any random edge is added). so that routing is not trivial anymore. Fortunately an elegant solution does exist (see next section). The proposed solution also automatically avoids routes through congested servers and links. The routing information is updated dynamically without human intervention.

3.2 state information

To minimize memory usage of a server and link bandwidth, we must only send what is really needed. If, for example, a server does not have any users on a certain channel, it does not have

to know what the channel characteristics are, or what users are on the channel. Only when a user requests this information from the server, the server must try to obtain this information. So we must create some sort of channel-membership for servers. They must subscribe to a channel first, after which they receive all data necessary to maintain the channel data locally. Servers also do not need to know all users. They only know the users that are participating in the channels that the server is a member of. This also means that a server cannot verify if a nick is already in use. This is not a problem as long as the two users do not join the same channel. But then they can be identified using the serverid they're using; this serverid can be appended to their nicks. But any other arbitrary identification method can be used, such as unique digits or symbols. This also solves the problem IRC has with its nickname space: nicks are 9 characters only, and no identical nicks are allowed.

3.3 rejoining the net

If the network of conference servers should split and rejoin, no special action needs to be taken. When two servers encounter conflicting data, they start a short "discussion" as to what version of the data should be used. This means that all inconsistencies are slowly removed, so that no "net burst" is generated. This improves bandwidth efficiency and prevents link overloading. See also section 6 on Error Correction.

4. Structure of Chatnet

4.1 Network Objects

First, let us introduce the network objects:

Servers are identified using a unique 32-bit serverid. This id is given to them by a central authority. A server is either the entire protocol stack, or only the third layer.

Clients are identified by a 16-bit userid that is unique within the scope of the server. So the tuple (serverid, userid) uniquely identifies a user on the entire net.

Channels are identified using a case insensitive character string of at most 32 characters.

A **client** is a program that has connected to a server and identified itself as a client.

Terms that will be used to describe the protocol are the following:

SAP, or Service Access Point: an abstract point at which interaction between a service provider and a service user takes place.

Service Primitive: a Service Primitive describes the interaction that may take place. It can have parameters, for example ConnReq (address). This Service Primitive can indicate, for example, that the service user requests the service provider that a connection be made to "address".

Service User: an abstract entity that can use a service offered at a SAP.

Service Provider: an abstract entity that can offer a service at a SAP.

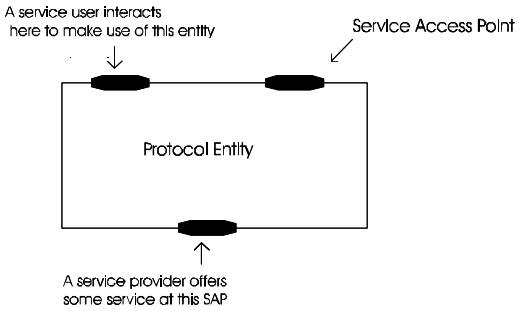

Fig. 1: graphical representation of a protocol entity

Protocol Entity: an abstract entity that implements a protocol. It usually offers some service at the "top" when modelled graphically (see fig.1), and makes use of some service at the bottom.

Protocol Information: the information kept by a protocol entity.

Peer entitity: a protocol entity with the same functionality, but residing in a different computing environment (another computer system, or merely another process on the same system).

A graphical representation of the Chatnet protocol structure is given in fig. 2. The three layers will be treated below.

4.2 The first layer: Network Interface Layer

The protocol is designed such that it is not dependent of the underlying network service. Any reliable, connection oriented network can be used to transport data between protocol entities. In order to provide a uniform interface, a Network Interface Layer is used. This layer offers the following functionality to the service user:

- establishing a connection

- identifying and authenticating servers and clients
- service negotiation (very rudimentary)
- buffering and transfer of data
- connection termination.

The Network Interface Layer must provide a unique connection identifier for each existing connection, in other terms: a Service Access Point for each connection. This can then be used by the service user to identify a certain connection; the service user is not bothered by network addresses, transport protocols etcetera. It only notifies the service user of incoming connections *after* they have been authorized. An incoming connection identifies its "type" (client or server), its username or serverid, its protocol version, software version and a password. This password is checked against a database. If it is wrong, the connection is terminated; otherwise, the appropriate service user is notified of the new connection (for each type, a different service user may be informed). If the server cannot handle more connections or is not willing to, it may send a FULL message and terminate the connection. The NIL layer furthermore handles aliveness of communication links using PING messages (to which the other end must respond within limited time using a PONG message). Connections between servers are also called "links". The other protocol information that the NIL layer must maintain is the following: for each existing connection, it must store the connection id that this connection has been given. It must furthermore store characteristics like network address of the other end, the software running there, the protocol version that must be used to communicate with the other end, etcetera. In order to identify the other end, it must also store the serverid if it is a server, or a unique userid created by the NIL layer if it is a client.

The service primitives that are provided are the following:

```
nil_server_dataind (connid, data)
nil_server_discind (connid, serverid)
nil_server_connind (connid)
nil_client_connind (connid)
nil_client_dataind (connid, data)
nil_client_discind (connid)
```

The names quite explain what the service primitives do. These primitives are the only way for a service user to interact with the NIL layer. Since the points of interaction are limited, the influences of the service user are clear.

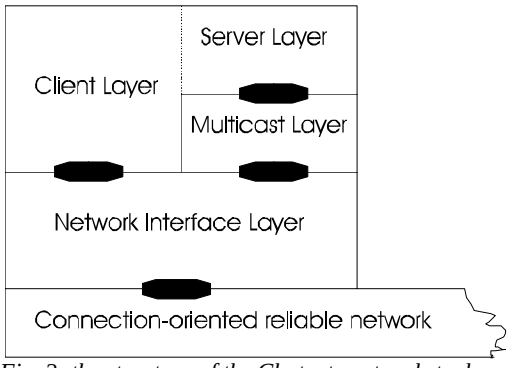

Fig. 2: the structure of the Chatnet protocol stack

4.3 The second layer: Multicast Layer

Above the NIL layer is the Multicast layer and the Client layer. Thus the Multicast Layer is a service user of the NIL layer. It is only used by servers; clients talk directly to the NIL layer. The multicast layer provides the following functionality:

- transmission of messages to a list of destination serverid's
- broadcasting of messages to all servers that can be reached
- routing of messages to get them to their destination(s)
- resequencing of messages
- maintaining routing information
- building spanning trees with different roots (future extension)
- gathering global characteristics from all nodes in the entire graph (future extension)

The algorithms to build the routing information are explained in the next section.

The Multicast layer uses datagrams. A datagram is a message that, amongst others, contains a source identifier, a destination identifier and the message body. The destination identifier can be a single destination, or a list of destinations.

4.3.1 The routing information

This consists of the following: for each server, a list of links is maintained, in the order of response times. Since servers are identified using a unique serverid, this id may be used as an index to an array of a list of link id's (as a link is a connection between two servers, the link id is just the connection id). Furthermore, the Multicast

layer must maintain a list of all links it has to other servers. It must know for each link at what rate it may send data on that link. If, for example, a particular server is allowed to connect via a serial link, then the NIL layer must know in what rate it may send data via this link. Or, suppose that the server is only allowed to use a limited bandwidth on a particular network, it may be specified that only 10kbit/s may be used. This information will typically be provided in a configuration file.

The layer also maintains a list of recently seen broadcasts. This list may be cleaned at regular intervals, removing old broadcasts. The interval depends on the maximum delay in the network; care must be taken that no broadcast is removed from the list while one or more copies of it could be underway.

4.3.2 Message routing

The way messages are routed using this information is as follows:

Multicast messages: The protocol entity sorts the serverid's in the destination list on the link they can be reached fastest. If the datarate is exceeded, it must use the second-fastest link, etcetera. If the datarate of all links is exceeded, it must send the remaining data on the supposedly fastest link. The server administrator is preferedly notified of this event, as it is an indication that the server hasn't got enough resources to fulfill its task. After the list has been sorted into smaller lists, the message is sent over all links that have a non-empty destination list. Say, for example, that server 1 wishes to send a message to servers 2, 4 and 15 (see fig. 3). Suppose that server 4 and 15 can be reached fastest via link 5, and that server 2 can be reached via link 3. The Multicast layer then creates a destination list consisting of servers 4 and 15, and sends this destination list along with the message to link 5. It also creates a destination list with only server 2 and sends it to link 3. Note that it may very well be that it is not server 2 itself that is directly connected via link 3; the message will be passed on from server to server until it reaches its destination. If a server receives a multicast message on a link, it treats it as if it sent it itself (sorting, copying and dispatching the message). There is one but: the Multicast layer must check that no information is sent back over the link the message arrived on; this way, some protection is offered against cycling messages.

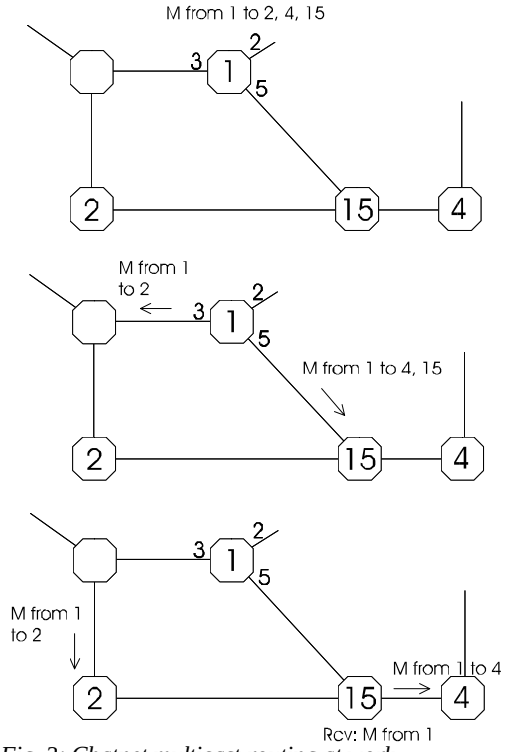

Fig. 3: Chatnet multicast routing at work

Unicast messages Unicast messages are messages destined for one server. As may be deduced from the explanation above, a unicast message is merely a multicast message with a destination list consisting of only one server. It may thus be treated as a multicast message.

Broadcast messages: Broadcast messages are simply sent across every link that the Multicast layer has to other servers. This may seem very expensive, but broadcasting ("flooding") is part of the routing mechanism as will be explained in the next section. If a server receives a broadcast from say server S, it examines the sequence number N attached to it. If it has already received a broadcast from S with number N, it updates its routing information. If it has not yet seen this broadcast, it sends it over every link it has to other servers, except the link it arrived on.

Bounced messages: When a particular server A does not have a link to a server D, it must "bounce" the message. This means that it is marked as undeliverable, and sent over the link towards the source server S of the message, say to server B. When server B receives the bounced message on link x, it removes link x from the list of links for server D. Doing so, it remembers the fact that server D could not be reached via link x.

Server B then looks for another link to send the message on towards server D. If it finds one, it sends the message across this link, removing the "undeliverable" mark. If it does not have a link to server D, it bounces the message further back towards its source S. So the message wanders through the network, updating routing information on the way, until either:

- it reaches its destination D;
- it reaches its source S; S now knows that server D cannot be reached. In this case the graph must be unconnected. It notifies the service user of the fact that server D is unreachable.
- some server doesn't have a link either to the destination D or the source S. The message is then said to be "trapped" in a network partition where nor its source, nor its destination is in. The message may then be discarded. (Note that this is the only case in which a message may be discarded by a server; messages can further be lost when a server crashes while the message is in its buffer).

4.3.3 Service primitives

The following service primitives are offered:

```
mcast_dataind (serverid, data)
mcast_unreachable (serverid)
mcast_datareq (destlist, data)
mcast_connreq (serverid)
mcast_broadcast (data)
```

Again, the names will speak for themselves.

4.3.4 message resequencing

As messages from a certain source may travel via different routes, they may become reordered by the network. The Multicast layer must therefore restore the order of all messages that arrive. It may do this by maintaining a buffer of a certain length for each server it is currently receiving data from, to store messages in-order. As "holes" are filled, it may send the messages to the service user. There must also be a timeout mechanism on every waiting hole, because messages can (in rare cases) get lost. In this case, the Multicast layer must forget about the message and send what it's got to the service user. State information will repair itself (see section 6), but the loss of conference data will not be noticed. Due to the severity of this failure, the server administrator must be informed about this event.

4.3.5 future extensions

Two mechanisms have already been defined but not yet incorporated into the protocol: finding a spanning tree (the algorithm finds a spanning tree using l+t messages, where l is the number of links in the graph, and t the number of links in the tree). This algorithm will be used to support report channels. These are channels with only a few senders and a very large group of receivers (named after #report on IRC at the time of the gulf war). The second is an algorithm that collects linear information. Linear information is information that has a sum operator defined on it, so that a piece of information can be added to a "sub total" at each server. This algorithm could be used to determine the total numbers of users on-line throughout the net, or a numeric representation of the total graph of the net. The algorithm takes l+t messages to return the "sum" to the server that initiated the algorithm.

4.4 The third layer: Chatnet Server Layer

This layer maintains the actual conference information base. It interacts with the Multicast layer in order to route inter-server traffic, and with the NIL layer to send data to clients. The Chatnet Server may actually be seen as consisting of a client-part and a server-part. Both parts share a large information base however. For this reason, we will keep viewing it as a single layer. The information that must be managed is the following:

For each user known to the server, a record must be maintained. In it is the current nickname, the user's real name, the user's network address, the server it is coming from, the userid, and the channels the user is on as far as this is known to the server. A list of all channels must be maintained. Creation and starvation of channels are broadcast through the net, so that the list of channels can be kept uptodate. Note that the name of the channel is the only attribute that the server knows.

For all channels the server is on, it must store a list of users that are on the channel, the channel characteristics (mode, topic etc) and all servers that are subscribed to the channel.

The server must also maintain a list of its own clients. The information is the same as the information for "foreign" users, except that the list is extended by a list of users the client is ignoring.

5. Chatnet routing

A Multicast layer obtains its routing information as follows: upon reception of the first copy of a broadcast with source S and sequence number N on a link x, it clears the list of links for this server, and places x at the head of the list. It stores the tuple (S, N) in a list so that it recognizes other copies of the broadcast. If another copy arrives at, say, link y, it adds y at the tail of the list. Thus a list of links is created in descending order of speed. As explained in the previous section, the link at the head of the list is used to transmit data on that is destined for server S, as this is the link that is the first step in a route that has proven to be fastest. Note that a Chatnet server does not know the exact route the information follows; it only knows the first hop. This scheme avoids congestion, as broadcasts travelling through congested areas will be delayed; they arrive much later than broadcasts that travelled another route, so that the congested area is avoided. It is not known yet if "oscillation" will occur. Oscillation is said to happen if all servers decide at the same time that a certain route A is congested and switch to route B, with the effect that now route B will be congested and route A has no traffic anymore. After some time the servers will discover this and switch again, etcetera.

The list of links for each server that is obtained using broadcasts, can be modified on a number of occasions:

- reception of any multicast message. When a message from server S is received on link x, the Multicast layer checks if x is at the head of the list for S. If not, it removes x from the list and inserts it at the head. The Multicast layer will use link x if it has data for S. Doing so, it adapts to new routes that have been created one way or another. If, for example, somewhere in the graph two servers have established a much faster connection than other parts of the network, some traffic may be rerouted as other servers discover this faster path.

- reception of a bounced message. As mentioned before, reception of a bounced message on link x destined for server D causes x to be removed from the list of server D.

- link breaks. When a network connection breaks for some reason, the NIL layer will send a DiscInd(Cid) to the Multicast layer. The Multicast layer must then search through the list of ALL servers, removing link Cid from the servers' list.

- link creation. Servers may create links arbitrarely. If server A creates a link x with server B, both put x at the head of the list for the other server. Note that x may have different values at both ends, as the domain in which links are numbered, is limited to the memory of each server. The existence of this link is slowly propagated through the graph; let's say server C has a link y with B, and sends its traffic to A via another link, z. It then notices that traffic for server A is suddenly coming from link y, as server B has a direct link to A. According to the rules explained above, server C puts link y at the head of the list for server A, so that it will now use the more direct route to A.

6. Error correction

The protocol has a number of provisions to correct errors. Errors can be divided into two classes: routing errors and conference information inconsistencies. As explained in section 4 and 5, routing errors are resolved using the concept of "bounced messages". The mechanism to correct conference information errors works about the same: a server A noticing a message that is inconsistent with its own informationbase, sends it back to where it came from, say server S. Note that it may not "bounce" it using a Multicast bounce, nor must it use the same sequence number. It is a totally new message with a life of its own. Upon reception of such an error message, server S examines the message and the error that was detected in it. It then prepares a new (set of) message(s) that should solve the inconsistency, sends these messages to A, and then resends the original message. If, for example, server S has a user U on channel C, and server A also has users on C. If U says a text T on C, a message goes to A saying "U said T on C". Now say that server A does not know that user U is on C. It then sends back "User not at channel: U said T on C". S then comes to the conclusion that it is quite sure that U is on C, so it sends to A the following message: "U joins C", and right after it, "U said T on C". A will then display a message to its clients that U joined C, and that U said T on C. The trick is to design the messages such that every inconsistency can be resolved. The proposed protocol includes messages for (hopefully) every possible

inconsistency that can be detected by servers. Note that due to the error-correcting properties, it is not necessary to exchange the state information between servers when the net is connected again after a partition. Instead of flooding the newly connected link with state information, every inconsistency is steadily removed piece by piece between individual servers. Although the message-rejection scheme causes extra messages to be sent, it still generates much less (bursty) traffic than IRC, as the inconsistencies are steadily recovered.

7. Current problems

7.1 scaling problems

There still are some things that don't scale very well. The following have been identified so far:

- A server needs to maintain a list of links for each server known in the net. Although it is not a lot of information (let's say a server allows max. 10 links to other servers, and it stores a link id in 1 byte, then it needs 10 bytes per server; in a network of five thousand servers this table takes 50 kilobytes), it still scales linearly with the number of servers in the net. The protocol can handle 2,1 billion servers, server memory can't.

- A server still needs to know all channels known throughout the net. Even worse, these channels need to be sent to new servers that just enter the net. The memory needed to store the channels also scales linearly to the number of users, as there is a "social" limit to the number of users per channel (30 to 40). So twice as much users will create twice as much channels, eating twice as much memory.

7.2 more hierarchy needed

- The channelname-space is flat. As the net grows, more channels are needed. However, an increasingly number of people might want to be in channels with a certain popular name. It would be a solution to allow several of these channels to be created with the same name, but in a different branche of the hierarchy tree; just like the usenet-news naming hierarchy.

- In order to achieve better routing and limit the number of servers a particular server has to have routing information for, it would be nice to allow different levels of servers, for example hubs which only route inter-server traffic between large "subnets". Hubs would maintain similar routing information about other hubs, and use the same mechanisms as the low-level servers do.

7.3 More distribution

Demanding that serverid's be obtained from a central authority isn't very neat. It would be more elegant if servers could decide amongst themselves who gets what id. It would however cost a lot of messages, and make servers a bit too anonymous. If a serverid is more "hardcoded", it is easier to trace server traffic and identify malicious or buggy servers.

8. Conclusion

Some major problems of IRC have been identified. Elegant solutions have been found, using a robust multicasting scheme. The remaining problems have also been identified. The proposed protocol is however only one of a dozen attempts to improve IRC or replace it by something better. Some attempts focus on patching IRC's networking problems, others try to design a more centralized protocol. Chatnet is a protocol that holds on to the distributed paradigm, providing a reliable worldwide conferencing system. Perhaps this is the long-waited for "irdc-three", the third issue of the IRC daemon.

And why not try to design such a conferencing protocol? Well consider this: how can one try to design large multimedia conferencing systems when there isn't even a highly scalable, worldwide text-based conferencing system?

9. References

[RFC1459] J. Oikarinen, "Internet Relay Chat Protocol", *Internet Requests for Comments*, No. 1459, Network Information Center, May 1993.