The Conference Control Channel Protocol (CCCP): A scalable base for building conference control applications

Mark Handley, Ian Wakeman and Jon Crowcroft
University College London
M.Handley@cs.ucl.ac.uk

Abstract

This paper presents the Conference Control Channel Protocol (CCCP), a new scheme intended for controlling conferences ranging from small, tightly coupled meetings, to extremely large loosely coupled seminars. We describe the requirements of such a scheme, and present a framework for building systems that connect together new and existing applications.

1 Introduction

In this paper, we will discuss some of the lessons that we have garnered from previous work involving computer based multimedia conferencing, and use these as a basis for developing an architecture for the next generation of conference control applications, suitable for conferencing over wide area networks. We show that a simple protocol acting over a conference specific communications channel, named the Conference Control Channel or CCC, will perform all tasks within the scope of conference control.

The previous generation of conferencing tools, such as CAR, mmconf, etherphone and the Touring Machine ([han], [cro], [sch], [vin] and [ara]), were based on centralised architectures, where a central application on a central machine acted as the repository for all information relating to the conference. Although simple to understand and simple to implement, this model proved to have a number of disadvantages, the most important of which was the disregard for the failure modes arising from conferencing over the wide area.

An alternative approach to the centralised model is the lightweight session model promoted by Van Jacobson and exemplified by the vat [cas],[ja1] and wb [ja2] applications. In the lightweight session model, connectivity is regarded as inherently unreliable. Our observations of the Mbone (see 2.1) show that humans can cope with a degree of inconsistency that arises from partitioned networks and lost messages, as long as the distributed state will tend to converge in time.

We have taken these and the other lessons we have derived from experience with conferencing tools, and derived two important aims that any conference control architecture should meet:-

1. The conference architecture should be flexible enough so that any mode of operation of the conference can be used and any application can be brought into use. The architecture should impose the minimum constraints on how an application is designed and implemented.
2. The architecture should be scalable, so that "reasonable" performance is achieved across conferences involving people in the same room, through to conferences spanning continents with different degrees of connectivity, and large numbers of participants. To support this aim, it is necessary explicitly to recognise the failure modes that can occur, and examine how they will affect the conference, and then attempt to design the architecture to minimise their impact.

We model a conference as composed of a (possibly unknown) number of people at geographically distributed sites, using a variety of applications. If an application shares information across remote sites, we distinguish between the cases when the participating processes are tightly coupled [1] - the application cannot run unless all processes are available and contactable - and when the participating processes are loosely coupled, in that the processes can run when some of the sites become unavailable. A tightly coupled application is considered to be a single instantiation spread over a number of sites,

[1] We define a tightly coupled system as one which attempts to ensure consistency at all sites. By contrast a more loosely coupled system tolerates inconsistencies, though it should attempt to resolve them in time. A very loosely coupled system will not even know the full list of conference members.

whilst loosely coupled applications have a number of (cooperating) instantiations.

The tasks of conference control break down in the following way:

- **Application control** - Applications need to be started with the correct initial state, and the knowledge of their existence must be propagated across all participating sites. Applications may need to cooperate (for example to achieve audio and video synchronisation).
- **Membership control** - Who is currently in the conference and has access to what applications.
- **Floor management** - Who or what has control over the input to particular applications.
- **Network management** - Requests to set up and tear down media connections between end-points (no matter whether they be analogue through a video switch, a request to set up an ATM virtual circuit, or using RSVP [zha] over the internet), and requests from the network to change bandwidth usage because of congestion.
- **Meta-conference management** - How to initiate and finish conferences, how to advertise their availability, and how to invite people to join.

We maintain that the problem of meta-conference management is outside the bounds of the conference control architecture, and should be addressed using tools such as LBL's Session Directory [ha4], traditional directory services or through external mechanisms such as email. The conference control system is intended to maintain consistency of state amongst the participants as far as is practical and not to address the social issues of how to bring people together, and co-ordinate initial information.

We then take these tasks as the basis for defining a set of simple protocols that work over a communication channel. We define a simple class hierarchy, with an application type as the parent class and subclasses of network manager, member and floor manager, and define generic protocols that are used to talk between these classes and the application class, and an inter-application announcement protocol. We derive the necessary characteristics of the protocol messages as reliable/unreliable and confirmed/unconfirmed (where 'unconfirmed' indicates whether responses saying "I heard you" come back, rather than indications of reliability).

We have abstracted a messaging channel, using a simple distributed inter-process communication system, providing reliable/unreliable semantics. The naming of sources and destinations is based upon application level naming, allowing wildcarding of fields such as instantiations (thus allowing messages to be sent to all instantiations of a particular type of application). Whether a message is confirmed or not is up to the application using the channel.

The final section of paper briefly describes the design of the high level components of the messaging channel. Mapping of the application level names to network level entities is performed using a distributed naming service, based upon multicast once again, and drawing upon the extensive experience already gained in the distributed operating systems field in designing highly available name services.

2 Requirements

2.1 Multicast Internet Conferencing

Since early 1992, a multicast virtual network has been constructed over the Internet. This multicast backbone or Mbone [mac] has been used for a number of applications including multimedia (audio, video and shared workspace) conferencing. These applications involved include vat (LBL's Visual Audio Tool), ivs (INRIA Videoconferencing System, [tur]), nv (Xerox's Network Video tool, [fre]) and wb (LBL's shared whiteboard) amongst others. These applications have a number of things in common:

- They are all based on IP Multicast.
- They all report who is present in a conference by occasional multicasting of session information.
- The different media are represented by separate applications [2]
- There is no conference control, other than each site deciding when and at what rate they send.

These applications are designed so that conferencing will scale effectively to large numbers of conferees. At the time of writing, they have been used to provide audio, video and shared whiteboard to conference with about 500 participants. Without multicast[3], this is clearly not possible. It is also clear that, with unreliable networks, these applications cannot achieve complete consistency between all participants, and so they do not

[2] Actually IVS does support audio, but has also been widely used as a pure video codec with vat as the audio tool.

[3] or broadcast (in the radio or TV sense), but that is outside the scope of this document

attempt to do so - the conference control they support usually consists of:

- Periodic (unreliable) multicast reports of receivers.
- The ability to locally mute a sender if you do not wish to hear or see them. However, in some cases stopping the transmission at the sender is actually what is required.

Thus any form of conference control that is to work with these applications should at least provide these basic facilities, and should also have scaling properties that are *no worse that the media applications themselves.*

The domains these applications have been applied to vary immensely. The same tools are used for small (say 20 participants), highly interactive conferences as for large (500 participants) disseminations of seminars, and the application developers are working towards being able to use these applications for "broadcasts" that scale towards millions of receivers.

It should be clear that any proposed conference control scheme should not restrict the applicability of the applications it controls, and therefore should not impose any single conference control policy. For example we would like to be able to use the same audio encoding engine (such as vat), irrespective of the size of the conference or the conference control scheme imposed. This leads us to the conclusion that *the media applications (audio, video, whiteboard, etc) should not provide any conference control facilities themselves, but should provide the handles for external conference control and whatever policy is suitable for the conference in question.*

2.2 The MICE Project Requirements

MICE is a European funded project to promote video conferencing for researchers. Since sites have a mix of equipment and capabilities, MICE is required to support:-

- Multicast based applications running on workstations.
- Hardware video and audio codecs and the need to multiplex their output.
- Sites connecting into conferences from ISDN.
- Interconnecting all the above.

These requirements have dictated that a number of multiplexing points are used to provide the necessary format conversion and multiplexing to interwork between the multicast workstation based domain and unicast (whether IP or ISDN) hardware based domain [ha2] and [ha3].

Traditionally such a multiplexing centre (or multipoint control unit) would employ a centralised conference control system such as the ITU's T.120 family of protocols, but on the Mbone that is not desirable as the reliable communication required by such an MCU would prohibit users from participating in very large multicast based conferences. It would also be inappropriate and inconvenient for the users to change multicast media applications when they switch from a entirely multicast based conference to one using a CMMC for some participants.

It is inevitable that translators, multiplexors, format converters and so forth will form some part of future conferences, and that large conferences will be primarily multicast based. Thus although CCCP has originated primarily from the needs of the MICE project, it has done so from a process of generalisation that should make it widely applicable.

2.3 Where current systems fail

The sort of conference control system we are addressing here cannot be:

- Centralised. This will not scale.
- Fixed Policy. This would restrict the applicability. The important point here is that only the users can know what the appropriate policies a meeting may need.
- Application Based. It is very likely that separate applications will be used for different media for the foreseeable future. We need to be able to switch media applications where appropriate. Basing the conference control in the applications prevents us changing policy simply for all applications.
- Homogeneous. Most existing systems have been fairly homogeneous. An increasing requirement is for different systems to interwork. There needs to be some basis for this interworking, at both the media data stream level and at the conference control level.

2.4 Specific requirements

Modularity: Conference Control mechanisms and Conference Control applications should be separated. The mechanism to control applications (mute, unmute, change video quality, start sending, stop sending, etc) should not be tied to any one conference control application in order to allow different conference control policies to be chosen depending on the conference domain. This suggests that a modular approach be taken, with for example, specific floor control modules being added when required (or possibly choosing a conference manager tool from a selection of them according to the conference).

A unified user interface: A general requirement of conferencing systems, at least for relatively small conferences, is that the participants need to know who is in the conference and who is active. *Vat* is a significant improvement over telephone audio conferences, in part because participants can see who is (potentially) listening and who is speaking. Similarly if the whiteboard program wb is being used effectively, the participants can see who is drawing at any time from the activity window. However, a participant in a conference using, say, vat (audio), ivs (video) and wb (whiteboard) has three separate sets of session information, and three places to look to see who is active.

Clearly any conference interface should provide a single set of session and activity information. A useful feature of these applications is the ability to "mute" (or hide or whatever) the local playout of a remote participant. Again, this should be possible from a single interface. Thus the conference control scheme should provide local inter-application communication, allowing the display of session information, and the selective muting of participants.

Taking this to its logical conclusion, the applications should only provide media specific features (such as volume or brightness controls), and all the rest of the conference control features should be provided through a conference control application.

Flexible floor control policies: Conferences come in all shapes and sizes. For some, having no floor control, with everyone sending audio when they wish, and sending video continuously is fine. For others, this is not satisfactory due to insufficient available bandwidth or a number of other reasons. It should be possible to provide floor control functionality, but the providers of audio, video and workspace applications should not specify which policy is to be used. Many different floor control policies can be envisaged. A few example scenarios are:

- Explicit chaired conference, with a chairperson deciding when someone can send audio and video.
- Audio triggered conferencing. No chairperson, no explicit floor control. Video data rate depends on who is sending audio.
- Audio triggered conferencing with a multiplexing point. Video streams are multiplexed depending on the audio channel activity.
- The traditional Mbone lightweight session model. No control.

Scaling from tightly coupled to loosely coupled conferences: CCCP originates in part as a result of experience gained from tightly coupled centralised systems, such as the CAR Multimedia Conference Control system [han] and also from Mbone based loosely coupled conferences. Tightly coupled conferences have advantages for small conferences where membership needs to be controlled. Loosely coupled conferences are the only way to achieve scalability, but the current lightweight sessions are too unrestricted for some uses.

CCCP allows the same media tools and same conference control *mechanism* to be used for both tightly and loosely coupled conferences, whilst allowing conference control *policy* to change.

3 CCCP

3.1 The Conference Control Channel

To bind the conference constituents together, a common communication channel is required, which offers facilities and services for the applications to send to each other. This is akin to the inter process communication facilities offered by the operating system. The conference communication channel should offer the necessary primitives upon which heterogeneous applications can talk to each other.

The first cut would appear to be a messaging service, which can support one-to-many communication, and with various levels of confirmation and reliability. We can then build the appropriate application protocols on top of this abstraction to allow the common functionality of conferences.

We need an abstraction to manage a loosely coupled distributed system, which can scale to as many parties as we want. Thus the underlying communication should use multicast. Many people have suggested that one way of thinking about multicast is as a multifrequency radio, in which one tunes into particular channels in which we are interested. We extend this model to build an Inter Process Communications model, on which we can build specific conference management protocols.

What do we actually want from the system?

- We want to ask for services
- We want to send requests to specific entities, or groups of entities and receive responses from some subset of them, with notifications sent out to others.

CCCP originates in the observation that *in a reliable network*, conference control would behave like a bus - addressed messages are put on the bus, and the relevant applications receive the messages, and if necessary respond. In the Internet, this model maps directly onto IP multicast. In fact the IP multicast groups concept is extremely close to what is required. In CCCP, applications have a tuple as their address: (instantiation, application type, address, conference id). We shall discuss exactly what goes into these fields in more detail later. In actual fact, an application can have a number of tuples as its address, depending on its multiple functions. Examples of the use of this would be:

Destination Tuple	Message
(1, audio, local, 1)	START_SENDING
(*, activity_mgmt, local, 1)	I_RECV <media> <address>
(*, session_mgmt, *, 1)	I_AM <name>
(*, session_mgmt, *, 1)	I_HAVE_MEDIA <media list>
(*, session_mgmt, *, 1)	USER_LIST <user list>
(*, floor_control, *, 1)	REQUEST_FLOOR
(1, floor_control, u@x.y.z, 1)	GRANT_FLOOR

and so on. The actual messages carried depend on the application type, and thus the protocol is easily extended by adding new application types.

3.2 CCC Names

Using this bus model, we based our naming scheme upon the attributes of an application that could be used in deciding whether to receive a message. We thus build a name tuple from three parts:

(instantiation, type, address, conf-id)

An application registers itself with its CCC library (and transparently through to a distributed nameserver in the more sophisticated CCC - more of that in a later version of this paper), specifying one or more tuples that it considers describe itself. In the current prototype design, a control group address or control host address or address list is specified at startup, and metaconferencing (ie, allocation and discovery of conference addresses) is assumed to be outside the scope the the CCC itself. The parts of the tuple are:

address In our model of a conference, applications are associated with a machine and optionally with a user at a particular machine. Thus we use a representation of the user or the machine as a field in the tuple, to allow us to specify applications by location.. The *address* field will normally be registered as one of the following:

- hostname
- username@hostname

When the application is associated with a user, such as a shared whiteboard, the username@hostname form is used, whereas applications which are not associated with a particular user, such as a video switch controller register simply as hostname. For simplicity, we use the domain naming scheme in our current implementation, although this does not preclude other identifiable naming schemes. Note that hostname matches only applications that are *not* associated with a user unlike *@hostname. When other applications wish to send a message to a destination group (a single application is a group of size 1), they can specify the *address* field as one of the following:

- username@hostname
- hostname
- *@hostname - This matches all applications on the given host.
- * - this is used to address applications regardless of location.

The CCC library is responsible for ensuring a suitable multicast group (or other means) is chosen to ensure that all possible matching applications are potentially reachable (though depending on the reliability mode, it does not necessarily ensure the message reaches them all).

It should be noted that in any tuple containing a wildcard (*) in the address, specifying the instantiation (as described below) does not guarantee a unique receiver, and so normally the instantiation should be wildcarded too.

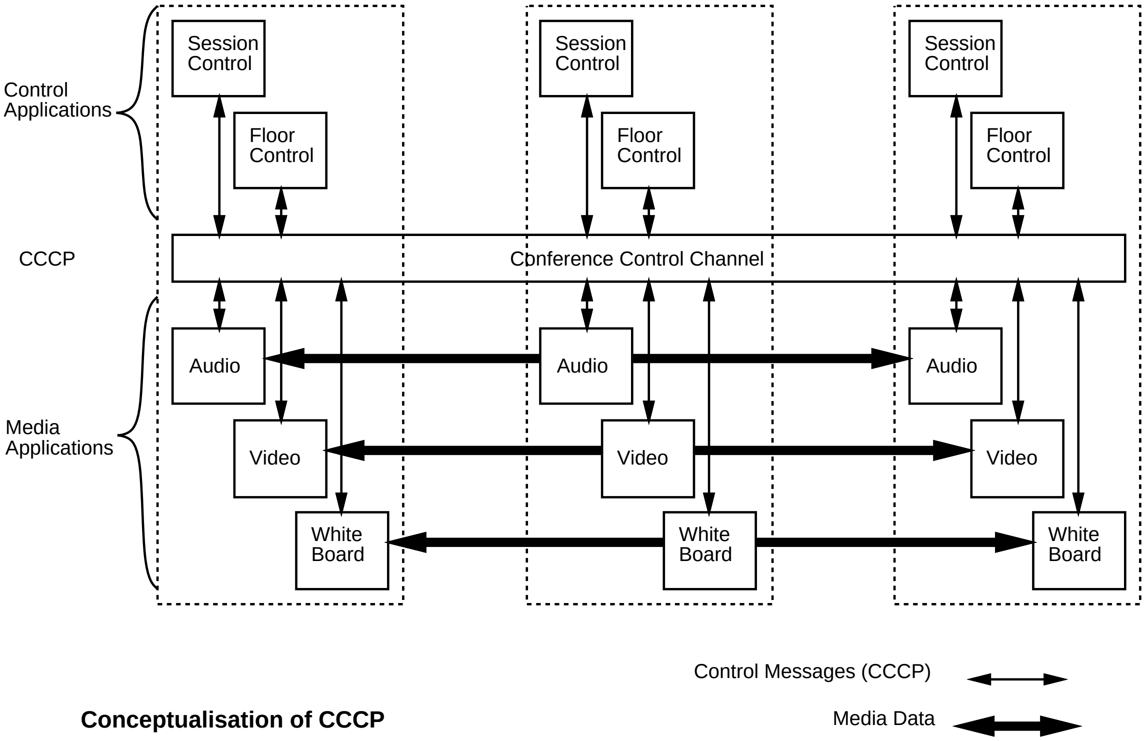

Figure 1: Conceptualisation of CCCP

type The primary attribute we use in naming applications is based on hierarchical typing of the application and of the management protocols. The type field is descriptive both of the protocol that is being used and of the state of the application within the protocol at a particular time. For example, a particular application such as vat may use a private protocol to communicate between instantiations of the application, so a vat type is defined, and only applications which believe they can understand the vat protocol and are interested in it would register themselves as being of type vat. An alternative way of using the type field is to embed the finite state machine corresponding to the protocol within the type field - thus a floor management protocol could use types floor.management.holder and floor.management.requester in a simple floor control protocol, that can cope with multiple requests at once. A final way of using the type field is to allow extensions to existing protocols in a transparent fashion, by simply extending the type field by using a version number. Some examples of these techniques can be found in the examples given.

Some base types are needed to ensure that common applications can communication with each other. As a first pass, the following types have been suggested:

- audio.send - the application is interested in messages about sending audio
- audio.recv - the application is interested in messages about receiving audio
- video.send - the application is interested in messages about sending video
- video.recv - the application is interested in messages about receiving video
- workspace - the application is a shared workspace application, such as a whiteboard
- session.remote - the application is interested in knowing the existence of remote applications (exactly which ones depends on the conference, and the session manager)
- session.local - the application is interested in knowing of the existence of local applications
- media-ctrl - the application is interested in being informed of any change in conference media state (such as unmuting of a microphone).
- floor.manager - the application is a floor manager

- floor.slave - the application is interested in being notified of any change in floor, but not (necessarily) in the negotiation process.

It should be noted that types can be hierarchical, so (for example) any message addressed to audio would be received by both audio.send and audio.recv applications. It should also be noted that an application expressing an interest in a type does not necessarily mean that the application has to be able to respond to all the functions that can be addressed to that type, although (if required) the CCC library will acknowledge receipt on behalf of the application.

Examples of the types existing applications would register under are:

- *vat* - vat, audio.send, audio.recv
- *ivs* - ivs, video.send, video.recv
- *nv* - nv, video.send, video.recv
- *wb* - wb, workspace
- a conference manager - confman, session.local, session.remote, media-ctrl, floor.slave
- a floor ctrl agent - flooragent, floor.manager, floor.slave

In the current implementation, the type field is text based, so that debugging is simpler, and we can extend the type hierarchy without difficulty.

instantiation The instantiation field is purely to enable a message to be addressed to a unique application. When an application registers, it does not specify the instantiation - rather this is returned by the CCC library such that it is unique for the specified *type* at the specified *address*. It is not guaranteed to be globally unique - global uniqueness is only guaranteed by the triple of (instantiation, type, address) with no wildcards in any field. When an application sends a message, it uses one of its unique triples as the source address. Which one it chooses should depend on to whom the message was addressed.

conference id The conference identifier is useful in unifying conference management and conference meta-management, and considerably simplifies the design of applications which may be part of multiple conferences simultaneously. It should be supplied to conference tools at startup, or alternatively supplied to media tools over the CCC. A default conference id is supplied for tools which are not currently involved in a conference. We omit conf-id from the examples later for simplicity.

3.3 Reliability

CCCP would be of very little use if it were merely the simple protocol described above due to the inherent unreliable nature of the Internet. Techniques for increasing the end-to-end reliability are well known and varied, and so will not be discussed here. However, it should be stressed that most (but not all) of the CCCP messages will be addressed to groups. Thus a number of enhanced reliability modes may be desired:

- None. Send and forget. (an example is session management messages in a loosely coupled system)
- At least one. The sending application wants to be sure that at least one member of the destination group received the message. (an example is a request floor message which would not be ACKed by anyone except the current floor holder).
- n out of m. The sending application wants to be sure that at least n members of the destination group received the message. For this to be useful, the application must have a fairly good idea of the destination group size. (an example may be joining of a semi-tightly coupled conference)
- all. The sending application wants to be sure that all members of the destination group received the message. (an example may be "join conference" in a very tightly coupled conference)

It makes little sense for applications requiring conference control to reimplement the schemes they require. As there are a limited number of these schemes, it makes sense to implement CCCP in a library, so an application can send a CCCP message with a requested reliability, without the application writer having to concern themselves with how CCCP sends the message(s). The underlying mechanism can then be optimised later for conditions that were not initially foreseen, without requiring a re-write of the application software.

There are a number of "reliable" multicast schemes available, such as [pet] and [bir], which can be used to build consensus and agreement protocols in asynchronous distributed systems. However, the use of full agreement protocols is seen to be currently limited to tightly coupled conferences, in which the membership is known, and the first design of the CCC library will not include reliable multicast support, although it may be added later as additional functionality.

We believe that sending a message with reliability "*all*" to an unknown group is undesirable. Even if CCCP can track or obtain the group membership transparently to the application through the existence of a distributed nameserver, we believe that the application should explicitly know who it was addressing the message to. It does not appear to be meaningful to need a message to get to all the members of a group if we can't find out who all those members are, since if the message fails to get to some members, the application can't sensibly cope with the failure. Thus we only support the *all* reliability mode to an explicit list of fully qualified (ie no wildcards) destinations. Applications such as joining a secure (and therefore externally anonymous) conference which requires voting can always send a message to the group with "at least one" reliability, and then an existing group member initiates a reliable vote, and returns the result to the new member.

3.4 Ordering

Of course loss is not the only reliability issue. Messages from a single source may be reordered or duplicated and due to differing delays, messages from different sources may arrive in "incorrect" order. There are many possible schemes for ordering messages from a single source, almost all of which require a sequence number or a timestamp. Within CCCP, a source can ask for any of the following ordering constraints when registering to receive messages:

No constraint Pass the packets directly through to the application as they arrive. A suitable example is for session messages reporting presence in a conference.

Late discard Discard any packets that are older than the latest seen. Quite a number of applications may be able to operate effectively in this manner, such as informational event reporting of "join" and "leave" events.

Adaptive playout Using the timestamp in a message and the local clock, estimate the perceived delay from the packet being sourced that allows 90% of packets to arrive. When a packet arrives out of order, buffer it for this delay minus the perceived trip time to give the missing packet(s) time to arrive. If a packet arrives after this timeout, discard it. A similar adaptive playout buffer is used in vat for removal of audio jitter. This is useful where ordering of requests is necessary and where packet loss can be tolerated, but where delay should be bounded.

Fixed playout Similar to above, specify a fixed maximum delay above minimum perceived trip time, before deciding that a packet really has been lost. If a packet arrives after this time, discard it.

Adaptive playout with ceiling A combination of [3] and [4]. Some delay patterns may be so odd that they upset the running estimate in [3]. Many conference control functions fall into this category, ie time bounded, but tolerant of loss.

ARQ Explicitly acknowledge every packet, allowing the sender to use timeouts to govern retransmission.

It should be noted that all except *No Constraint* require state to be held in a receiver for every source. As not every message from a particular source will be received at a particular receiver due to CCCP's multiple destination group model, receiver based mechanisms requiring knowing whether a packet has been lost will not work unless the source and receiver use a different sequence space for every (source, destination group) pair. Since we wish to avoid this, we must restrict the application level ARQ to a window size of 1. If the sender is attempting a reliable transmission, then it will use the underlying transport protocol (such as TCP) to maintain the state.

Although we believe these ordering mechanisms to be adequate for most cases, an application (with some semantic knowledge) can build a more elaborate mechanism on top of any of the ordering constraints. However it does mean that *message timestamps and sequence numbers must be available at the application level.*

CCCP does not intend to provide clock synchronisation and global ordering facilities, since this increases complexity and the necessary state within CCCP. If applications require this, they must do so themselves. However, for most applications, a less complex option is to design the application protocol to tolerate temporary inconsistencies, and to ensure that these inconsistencies are resolved in a finite number of exchanges. An example is the algorithm for managing shared teleconferencing state proposed by Scott Shenker, Abel Weinrib and Eve Schooler [she].

For algorithms that do require global ordering and clock synchronisation, CCCP will pass the sequence numbers and timestamps of messages through to the application. It is then up to the application to implement the desired global ordering algorithm and/or clock synchronisation scheme using one of the available protocols and algorithms such as NTP [mil] or ordering protocols such as [lam], [bir], or [pet].

3.5 A few examples

Before we describe what should comprise CCCP, we will present a few simple examples of CCCP in action. There are a number of ways each of these could be done - this section is *not* meant to imply these are the best ways of implementing the examples over CCCP.

3.5.1 Unifying user interfaces - session messages in a "small" conference

This requirement is described in section 2.4.

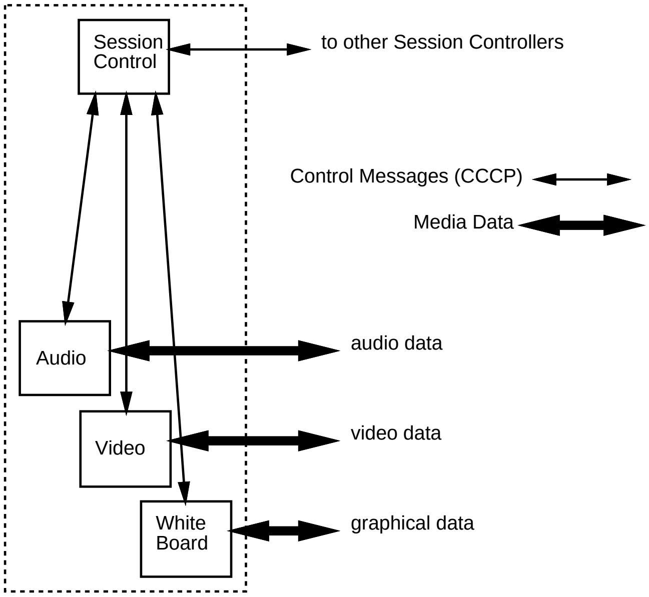

Figure 2: Using CCCP to unify interfaces

Applications:

- An Audio Tool (at), registers as types: at, audio.send, audio.recv
- A Video Tool (vt), registers as types: vt, video.send, video.recv
- A Whiteboard (wb), registers as types: wb, workspace
- A Session Manager (sm), registers as types: sm, session.local, session.remote

The local hostname is x. There are a number of remote hosts, including y and z.

A typical exchange of messages may be as follows:

From	To	Message
the following will be sent periodically:		
(1,audio.recv,x)	(*,sm.local,x)	KEEPALIVE
(1,video.recv,x)	(*,sm.local,x)	KEEPALIVE
(1,wb,x)	(*,sm.local,x)	KEEPALIVE
the following will be sent periodically with interval		
(1,sm,x)	(*,sm.remote,*)	I_AM <user name>
(1,sm,x)	(*,sm.remote,*)	I_HAVE_MEDIA audio.recv video.recv wb
an audio speech burst arrives at the audio tool from y		
(1,audio.recv,x)	(*,sm.local,x)	MEDIA_STARTED audio y
session manager highlights the name of the person speaking		
speech burst finishes		
(1,audio.recv,x)	(*,sm.local,x)	MEDIA_STOPPED audio y
session manager de-highlights the person who stopped speaking		
video starts from z		
(1,video.recv,x)	(*,sm.local,x)	MEDIA_STARTED video z
periodical reports:		
(1,audio.recv,x)	(*,sm.local,x)	KEEPALIVE
(1,video.recv,x)	(*,sm.local,x)	MEDIA_ACTIVE video z
(1,wb,x)	(*,sm.local,x)	KEEPALIVE
someone restarts the session manager:		
(1,sm,x)	(*,*,x)	WHOS_THERE
(1,audio.recv,x)	(*,sm.local,x)	KEEPALIVE
(1,video.recv,x)	(*,sm.local,x)	MEDIA_ACTIVE video z
(1,wb,x)	(*,sm.local,x)	KEEPALIVE
and so on...		

This example assumes that the session managers are responsible for communicating the participants of the conference to other sites. In practice, most of the media tools in such a conference would be likely to use the Real-time Transport Protocol (RTPv2). This specifies that applications should send session messages (RTCP messages) which communicate the participants for a particular *media*. In such cases, there is little point in duplicating this traffic with equivalent messages between session managers, so instead the applications would pass on the participant information they derive from RTCP messages to the local session manager, which then collates and presents this information to the user. Thus there is then no need for direct communication between session managers.

3.5.2 A voice controlled video conference

In this example, the desired behaviour for participants to be able to speak when they wish. A user's video application should start sending video when their audio application starts sending audio. No two video applications should aim to be sending at the same time, although some transient overlap can be tolerated.

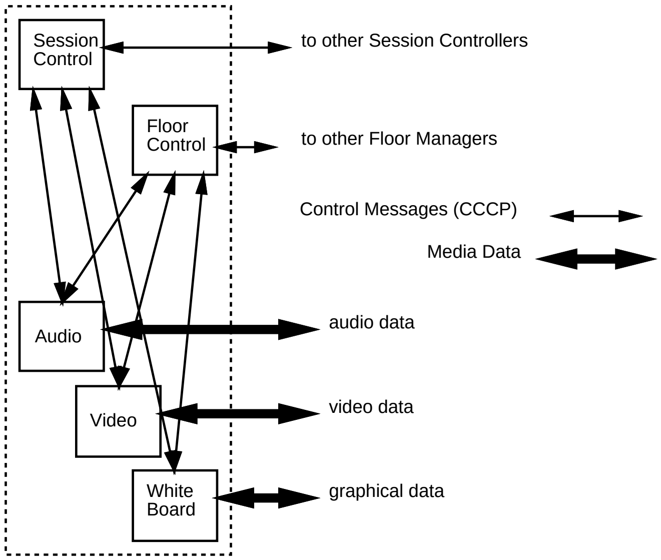

Figure 3: Using CCCP for simple floor control

Applications:

- An Audio Tool (at), registers as types: at, audio.send, audio.recv
- A Video Tool (vt), registers as types: vt, video.send, video.recv
- A Session Manager (sm), registers as types: sm, session.local, session.remote
- A Floor Manager (fm), registers as types: fm, floor.master

There are hosts x and y, amongst others. It is assumed that session control messages are being sent, as in the example above. The message exchange can be seen in Figure 4

3.6 CCCP Messages

CCCP messages have a plain text addressing scheme and a plain text payload. The addressing scheme consists of a source tuple (as described in section 3.2) which identifies which module in a conference originates the message and a list of destination tuples to which the message should be delivered. In addition there are a number of header bits which specify the reliability mode.

The payload can be free format with the exception that it must start with a function name which identifies the purpose of the message. The function name is normally followed by a space separated parameter list, but it is up to the receiving application to parse this.

(SRC tuple)<list of (DST tuple)s> FUNCTION <parameter list>

3.7 More complex needs

3.7.1 Dynamic type-group membership

Many potential applications require to be able to contact a server or a token holder reliably without necessarily knowing the location of that server. An example may be a request for the floor in a conference with one roaming floor holder. The application requires that the message gets to the floor holder if it is at all possible, which may require retransmission and will require acknowledgment from the remote server, but the application writer should not have to write the re-transmission code for each new application. CCCP supports "at least one" reliability, but to address such a REQUEST_FLOOR message to all floor managers is meaningless. By supporting dynamic type-groups CCCP can let the application writer address a message to a group which is expected to have only one (or a very small number) of members, but whose membership is changing constantly.

In the example described, the application requiring the floor sends:

SRC Tuple	DEST Tuple	Message
(1,floor.master,x)	(*,floor.master.holder,*)	REQUEST_FLOOR

with "at least one" reliability. Retransmissions continue until the message is acknowledged or a timeout occurs.

When the floor holder receives this message, it can then either send a grant floor or a deny floor message:

(1, floor.master, y) (1, floor.master, x) GRANT_FLOOR

This message is sent reliably (ie, retransmitted by CCCP until an ACK is received).

On receiving the GRANT_FLOOR message, the floor manager at x expresses an interest in the type-group

From	**To**	**Message**
the user at x starts speaking. Silence suppression cuts out, and the audio tool starts sending audio data:		
(1, audio.send, x)	(*,sm.local,x),(*,floor.master,x)	MEDIA_STARTED audio x
...this causes the sm to highlight the "you are sending audio" icon. It also causes the floor manager to report to the other floor managers:		
(1, floor.master,x)	(*, floor.master, *)	MEDIA_STARTED audio x
and also it requests the local video tool to send video:		
(1, floor.master,x)	(*, video.send, x)	START_SENDING video
...this causes the video tool to start sending		
(1, video.send, x)	(*, sm.local, x),(*.floor.master, x)	MEDIA_STARTED video x
...which, in turn, causes the sm to highlight the "you are sending video"icon		
the user at x stops speaking. Silence suppression cuts in, , and the audio tool stops sending audio data		
(1, audio.send, x)	(*,sm.local,x),(*,floor.master,x)	MEDIA_STOPPED audio x
...this causes the sm to de-highlight the "you are sending audio" icon. The session manager starts a timeout procedure before it will stop sending video		
...		
a user at y starts sending audio and video data. The local audio and video tools report this to the session manager:		
(1,audio.recv,x)	(*,sm.local,x)	MEDIA_STARTED audio y
(1,video.recv,x)	(*,sm.local,x)	MEDIA_STARTED video y
...as in previous example, sm highlights sender's name. Also y's floor manager reports what's happening		
(1, floor.master, y)	(*, floor.master,*)	MEDIA_STARTED audio y
(1, floor.master, y)	(*, floor.master,*)	MEDIA_STARTED video y
the local floor manager tells the local video tool to stop sending		
(1, floor.master,x)	(*, video.send, x)	STOP_SENDING video
...this causes the video tool at x to stop sending		
(1, audio.send, x)	(*,sm.local,x),(*,floor.master,x)	MEDIA_STOPPED video x
...		

Figure 4: Message exchange for a voice controlled video conference

floor.master.holder. On sending the GRANT_FLOOR message, the floor manager at y also removes its interest in the type-group floor.master.holder to prevent spurious acking of other REQUEST_FLOOR messages. However, if the GRANT_FLOOR message retransmissions time out, it should re-express an interest.

See section 3.8 on the Naming Service for more details of how dynamic type-groups work.

3.7.2 Need to know

When an application sends a message, it is up to the sending application to choose the reliability mode for the message. For example, in a large loosely coupled conference, a floor change announcement may be multicast in an unreliable mode. However, there may be a number of applications that really do require to see that information. In the floor control example, the existing floor holding applications need to see the floor change announcements. We propose allowing a receiving application to modify the reliability with which other applications send specific messages by allowing messages of the form:

(SRC tuple) (*, floor.manager, *) NEED_TO_KNOW
(*, floor.slave, *) <list of fns>

In this case the application specified by the source tuple is telling all floor.manager applications that when they send one of the specified functions to (*, floor.slave, *), this application would like reliable delivery of the message.

NEED_TO_KNOW messages should be sent periodically, and will timeout if one hasn't been received in a set amount of time. NEED_TO_KNOW requests will also time out at a particular application if that application ever fails to reliably deliver a message to the specified address. Clearly NEED_TO_KNOW messages should be used sparingly, as they adversely affect the scaling propertied of the CCC. However, there are a number of cases where they can be useful. The same effect could be achieved by declaring another type (for example floor.holder, which may be desirable in some cases), but NEED_TO_KNOW also has the benefit that it can be used to modify the behaviour of existing applications without a requirement to access the source code.

3.8 The Naming Service

CCC can be run on the bus model, passing all messages around a single multicast group per conference. This will scale reasonably, since it scales with the number of participants in the conference. Name resolution occurs at each host, matching the destination naming tuple in the message against the list of tuples that are registered at this particular host. However, it it does not scale indefinitely, because the load on each host and the network increases with the complexity of the conference and the number of messages. To improve scaling, the communications should be optimised so that messages are only propagated to the machines that are interested. Thus we need a service that maps the naming tuples to locations, so that intelligent mapping of message paths to locations can be performed (aka intelligent routing and placement of multicast groups). This name location service (or naming service as it is more generally known) has a number of properties that differentiate it from other naming services such as X.500 and the DNS:

- Dynamic and frequent updates.
- Fast propagation of changes.
- Ability to fall back to broadcast to interested parties when uncertain about the consistency of a refined addressing group, since names are unique per conference and are included in each message.

The last property is important, since it allows a relaxed approach to maintenance of consistency amongst the naming servers, saving greatly in the messages and complexity of the internals of the service.

We intend to implement a nameserver suitable for loosely coupled conferences as the default in the CCC library. However, CCCP will also allow the use of an external nameserver to supplement or replace the internal nameserver behaviour, which will allow much greater use of the nameserver to be made in more tightly coupled conferences, for instance by using the nameserver to keep an accurate list of members.

3.9 Security

The CCCP library currently supports encryption using DES, where all messages in a conference are encrypted using the same session key. Key distribution is through external means, and manual entry. Whilst this does provide a limited degree of confidentiality and protection, we plan to further investigate security for conferencing through the model offered by CCCP.

Stubblebine [stu] identifies scalable key distribution as a problem for conferences. Since access permissions may change for individuals during a conference, new keys may need to be distributed quickly and reliably to the recipients. We are investigating techniques for solving this problem using protocols built upon the CCCP architecture, based upon automatically configured hierarchies of trusted servers and public key cryptography.

Within a conference, the access and control rights over the various media and functions will vary across individuals and organisations. It is likely that the appropriate protection mechanisms will be a mixture of access control lists and encryption technology to ensure confidentiality, integrity and authentication. One approach is to limit access to the parts of the CCCP type hierarchy through the use of multiple keys. As the protection model becomes clearer, we will enhance CCCP to provide the necessary functionality, such as an authentication header, and experiment with security control protocols over CCCP.

3.10 Conference Membership Discovery

CCCP will support conference membership discovery by providing the necessary functions and types. However, the choice of discovery algorithm, loose or tight control of the conference membership and so forth, are not within the scope of CCCP itself. Instead these algorithms should be implemented in a Session Manager on top of the CCC.

4 Status and Future Work

The authors always had an implementation based on IP multicast in mind. However, every effort has been made to ensure there is nothing in CCCP that precludes implementation over unicast IP. However, CCCP does make the assumption the the Conference Communication Channel (however implemented) is always available. On systems based over circuit switched channels such as ISDN, this may not be the case.

We have implemented two basic versions of a CCCP library for testing and evaluation purposes, and are in the process of writing or modifying conference tools to utilise these libraries. These libraries currently use a single multicast address per CCC, but scope local messages appropriately.

An interesting area we hope to look at is the binding of multicast addresses to the application level names, ranging from imposing application level semantics upon the addresses through binding based solely on the measured traffic characteristics.

5 Acknowledgments

We would like to thank Isidor Kouvelas, Stuart Clayman, Atanu Ghosh, Dave Lewis and Thierry Turletti for their valuable comments and suggestions. This work is funded by the European Commission under the MICE ESPRIT 7602 and PREPARE RACE 2004 projects.

6 References

[ara] M. Arango et al: Touring Machine: A software platform for distributed multimedia applications, IFIP Upper Layer Protocols, Architectures and Applications, Vancouver Canada, June 1992

[bir] Birman K. P., Schiper, A. and Stephenson P. (1991), Lightweight Causal and Atomic Group Multicast, ACM Transactions on Computer Systems 9(3), 272-314

[cas] S. Casner: First IETF Internet Audiocast, ACM SIGCOMM Computer Communication Review, 22, pp92-97, July 1992

[cro] T. Crowley et al: MMConf: An infrastructure for building Shared Multimedia Applications, Proceedings of CSCW '90, Los Angeles, USA, October 1990

[fre] Ron Frederick: nv, UNIX Manual Pages, Xerox Palo Alto Research Centre

[han] Handley, MJ et al: Multimedia conferencing: from prototype to National pilot, Proc INET '92, pp 483-490, Internet Society, Reston, VA, USA, 1992.

[ha2] Handley, MJ and Clayman, S: Specification of the MICE Conference Management and Multiplexing Centre, MICE Project Internal Report, Jan 1994.
ftp://cs.ucl.ac.uk/mice/reports/draft-cmmc-spec-2.ps.Z

[ha3] M. Handley, P. Kirstein, A. Sasse: Multimedia Integrated Conferencing for European Researchers (MICE): Piloting Activities and the Conference Management and Multiplexing Centre, Computer Networks and ISDN Systems, V 26, pp275-290, Nov 1993

[ha4] Mark Handley, Van Jacobson: SDP - Session Description Protocol, Internet Draft, March 1995.

[kou] Isidor Kouvelas et al - A naming service for the CCC (under preparation)

[ja1] Van Jacobson and Steve McCanne: vat, UNIX Manual Pages, Lawrence Berkeley Laboratory, Berkeley, CA.

[ja2] Van Jacobson and Steve McCanne: Using the LBL Network Whiteboard, Lawrence Berkeley Laboratory, Berkeley, CA.

[lam] Leslie Lamport, Time, Clocks and the Ordering of Events in a Distributed System, Comm. of the ACM, V 21, pp558-565, July 1978

[mac] Mike Macedonia and Don Brutzman: MBONE provides Audio and Video across the Internet, IEEE Computer, v27,4, April 1994

[mil] D. Mills: Network Time Protocol (v3), RFC1305, September 1992

[pet] Peterson, L. L., Bucholz, N. C., and Schlichting, R. D. (1989) Preserving and Using Context Information in Interprocess Communications, ACM Transactions on Computing Systems, 7,(3), 217-246

[sch] Eve Schooler: The Connection Control Protocol: Architecture Overview ISI/RS-92-294 January 1992

[she] Scott Shenker, Abel Weinrib and Eve Schooler: An Algorithm for Managing Shared Teleconferencing State, Internet Draft, Oct 1993

[stu] Suart Stubblebine: Security Services for Multimedia Conferencing, Proc. of the 16th National Computer Security Conference, Baltimore, Ma, September 1993.

[tur] H.261 Software Codec for Videoconferencing over the Internet, Research Report No 1834, INRIA, Sophia-Antipolis, France, January 1993

[vin] Harrick M. Vin et al: Multimedia Conferencing in the Etherphone Environment, IEEE Computer, v24,10, October 1991

[xse] Xsecurity, X11 Release 5 manual page, Massachusetts Institute of Technology.

[zha] L. Zhang, S.Deering, D.Estrin, S.Schenker and D.Zappala: RSVP: A New Resource ReSerVation Protocol, IEEE Network, pp-8-18, September 1993.