Flexible Communication Support for CSCW Applications*

Hugo Miranda
Universidade de Lisboa
hmiranda@di.fc.ul.pt

Luís Rodrigues
Universidade de Lisboa
ler@di.fc.ul.pt

Abstract

Modern computer-supported cooperative work applications (CSCW) supporting same-time/different-place interaction are required to open several communication channels. Each of these channels has its own Quality of Service (QoS) and is implemented by a specific protocol stack. Typically, these channels need to be synchronized but inter-stack dependencies are hard to express with current communication architectures. The paper proposes a novel approach to the development of communication software supporting a style of micro-protocol composition that satisfies the requirements imposed by CSCW applications.

1 Introduction

The steady increase in communication bandwidth and processing power, and the ubiquity of the WEB presence, has brought same-time/different-place computer-supported cooperative work (CSCW) applications from the local-area network niche to the wide-area arena. Products such as the Microsoft NetMeeting [17] are helping to create the user demand for full-fledged CSCW applications that integrate in a seamless way audio, video, data and control channels.

On the other hand, wide-area networks pose significant challenges to the communication protocol designer. Common impairments such has the variability of network delays, message losses, network partitions, node crashes, among others, require the implementation of corrective measures that are media-specific. For instance, a video frame may be dropped (depending on the compression algorithm) but a resource lock request needs to be delivered in a reliable way. Additionally, these applications are typically multi-user, and this raises the need for group communication primitives that ensure a consistent view of critical application data.

As a result of these difficulties, complex CSCW application are often forced to manage different protocol stacks, each specialized for a given type of stream, such as audio, video, bulk-data and control information. Each of these stacks has been widely studied in isolation. Protocols tailored for video and audio have been proposed in [12, 6]. Protocols for data transfer can be found in [11]. Communication stacks that ensure strong semantics even in the presence of failures have been described in [18, 13]. However, the issue of coordinating these different stacks has been neglected until recently [2, 3, 5].

The paper proposes a communication architecture that allows different communication channels, each with its own QoS, to be integrated in a coherent multi-channel protocol stack. Furthermore, the architecture allows the application designer to specify the protocol stack that meets her/his QoS requirements through the composition of micro-protocols. A requirement constraining a single channel is called *intra-QoS* requirement. The integration of different media contents in a single application also imposes *inter-QoS* requirements. Video and audio, for example, must be synchronized at the receiver[1]. Previous

*Selected sections of this report were published in the Proceedings of the 5th International Workshop on Groupware, Cancun, Mexico, September, 1999.

[1] The SMIL protocol [8], for example, allows to define the

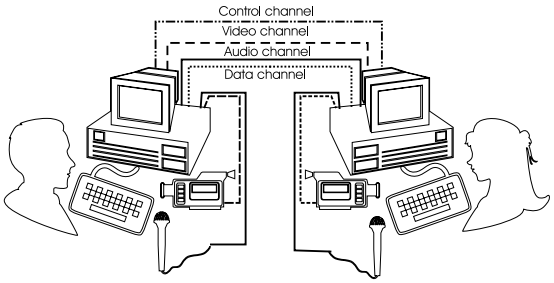

Figure 1: CSCW application

communication frameworks have limitations in satisfying inter-QoS requirements, delegating that burden to the application. We argue that providing inter-QoS properties at the communication level leads to a cleaner application design.

The paper is organized as follows. In section 2 we motivate the problem by presenting one example of the class of problems that impose inter-QoS constraints. Section 3 introduces related work in the are of CSCW and Group Communication Services. Section 4 describes a flexible communication architecture that addresses inter-QoS requirements. Section 5 concludes the paper.

2 Motivation

We motivate our work by giving an example from the class of problems that require the use of multiple QoS with both intra-QoS and inter-QoS constraints. The example is taken from [3]. It consists of a CSCW application where users share: i) video and audio; ii) a blackboard; iii) text, in the form of "chat" windows; iv) control information.

This application can be implemented using four different channels, as illustrated in Figure 1: an *audio* channel, a *video* channel, a *data* channel and a *control* channel. Each of these channels is supported by its own protocol stack and the following intra-stack QoS requirements can be identified:

- Messages on all channels must respect FIFO order.
- Audio and video channels can allow some extent of lost or garbled data.
- Text and control channels must be reliable.
- The control channel should provide strong semantics such as virtually synchronous totally ordered multicast [4]. This simplifies the management of the session control, including the management of resource locks.

In addition to these intra-sack requirements, several inter-stack constraints can also be listed:

- Audio and video frames should be synchronized with blackboard updates.
- Causal order should be preserved among control and text channels.
- Failure notifications should be provided in a consistent manner on all communication stacks (for an interesting discussion of the relevance of consistent failure detection in CSCW application see [7]).

In the next section, we will give an overview of how these constraints have been addressed in the literature.

3 Related Work

The Collaborative Computing Transport Layer (CCTL) [15] is a framework designed to support CSCW and multimedia applications. The authors address the need for supporting communication channels with different QoS requirements within a common session. CCTL presents a reliable control channel (the default channel) with strong properties (total order, FIFO, Failure detection and Name Service) and reliable or unreliable data channels providing total, FIFO or unordered delivery. Membership information flows through the control channel

application behaviour when synchronization problems arise.

and is processed by a *Channel Membership Submodule* that manages the membership of every data channel. Channels are not built using a modular framework, thus channels different from those pre-defined have to be built from scratch. The only *Inter-stack* constraint supported is the synchronization of membership changes. Inter-media data synchronization and ordering are not explicitly supported.

Maestro [3] is a flexible group manager for groups of protocol stacks. The base of Maestro is a core Ensemble [9] stack who handles membership on behalf of data stacks. Data stacks can be taken from a wide set of types, from UDP [14] sockets, to CCTL [15] and Ensemble stacks. The tool addresses problems such as the management of messages priorities and multilevel security. However, Maestro does not support *layer sharing* across two or more stacks. In this way, most inter-stack constraints other than membership have to be dealt directly by the application.

The Multimedia Multicast Transport Service (MMTS) [5] offers more flexibility in the management of multiple stacks. It integrates a Group Communication Service (Transis [1] or Horus [18]) with self-managed channels providing several QoS possibilities such as Unreliable FIFO and Reliable Unordered. While the group communication service channel provides membership and reliable control information, data channels transfer informations in "bunches", respecting marks imposed on the control channel. The "bunch" concept is a mechanism that allows the application designer to define, in a limited way, some inter-stack ordering constraints.

4 A Flexible Communication Framework

Protocol composition must be supported by a number of abstractions, tools and run-time mechanisms. Relevant run-time mechanisms include memory management for message structures, timeout management, thread management, etc. A classical example of work in this area is the *x*-kernel [10], which provides a powerful set of features to develop communication stacks. From the point of view of protocol composition, the function of the *protocol kernel* is to support the exchange of events between adjacent layers. This paper focus exclusively on the layer composition aspects of protocol kernels and does not addresses orthogonal issues such as buffer management, timer management, etc.

It should be noted that, in order to maintain layer independence, the micro-protocols should not be aware of the way they are interconnected. No explicit reference to a particular upper or lower layer should be allowed. Instead, each protocol should only invoke generic "up-event" and "down-event" calls. The events are delivered to the appropriate protocols by the kernel according to the bindings established when the stacks are created (possibly, at run-time). We propose a novel architecture that allows the routing of events to be based not only on QoS parameters (intra-QoS constraints) but also on inter-QoS constraints. The protocol kernel acts as a switching fabric, routing messages between layers and ensuring that a path satisfying the desired QoS is followed.

4.1 The model

We define a *layer* as the implementation of a protocol. All protocols implement the same *event interface*, which defines the types of events each one is able to consume and produce. The format and semantics of these events is irrelevant to our exposition. Typical examples of events are data transmission requests, indication and confirmations. Examples of layers are "datagram transport", "positive acknowledgment", "total order", "checksum", etc. Good examples of relevant layers and events in the context of fault-tolerant applications can be found in [9].

We define a *session* as an instance of a layer [10]. The session maintains state that is used by the layer code to process events. A layer that implements ordering may keep a sequence number or a vector clock as part of the session state. In connection oriented protocols, the session also maintains information about the endpoints of the connection. Note that it is often useful to maintain several active sessions for the same layer even when they share the same endpoints: for instance, one might want to have different FIFO channels for different priority streams.

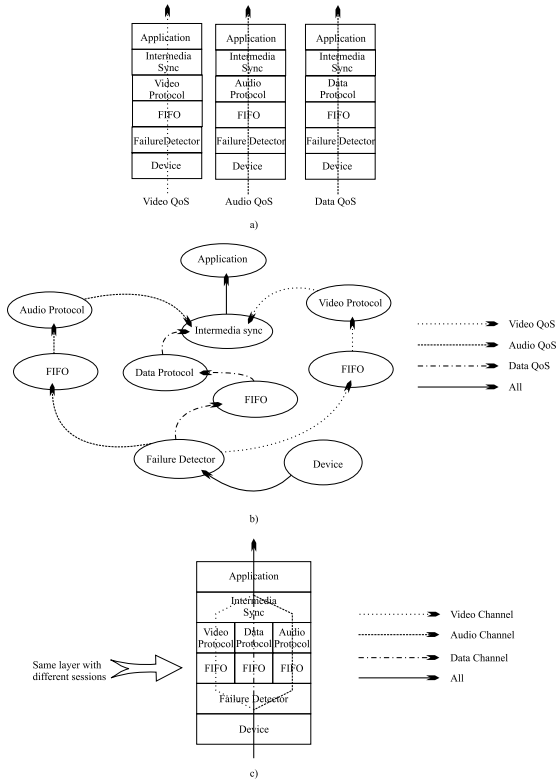

Figure 2: Inter-stack QoS requirements for a multimedia application in a) QoS set b) acyclic graph c) stack with three channels

The clear separation between layers and sessions is key to our design. These two concepts can be combined to satisfy multiple QoS requirements as follows:

A *QoS* is defined as an ordered set of layers. The QoS defines which protocols must act on the messages and by which order they must be traversed. A *channel* is an instantiation of a QoS and is [2] characterized by a stack of sessions of the corresponding layers. For sake of clarity we assume that all QoS that interact with the application are terminated on top by a layer that we simply call the "Application". A session of the Application layer is also called an

[2] At a finer level of granularity, one could also consider different parameterizations of each protocol as an independent QoS.

RQ-1 The user should be allowed to send a message using different Qualities of Service.

RQ-2 The message should follow peer channels in different nodes.

RQ-3 Channels may or may not share sessions of common layers. In particular, they may share the same endpoint.

RQ-4 Micro-protocols independence should be preserved.

Table 1: Implementation requisites

endpoint. A *stack* is a set of non-disjoint channels (i.e., each channel shares at least one session with some other channel of the same stack).

Every stack interfaces the application by at least one session of the APPLICATION layer. Each session of this layer serves as a serial access point to the stack: messages are sent to the lower layers by the order they are injected in the access point by the application. Similarly, messages are delivered to the application by the order they are received from the lower layers. The stack interfaces the communication media through a DEVICE layer. This layer is just an abstraction of any protocol outside the control of our communication kernel (for instance, TCP, UDP or MBONE). Figure 2 illustrates these concepts.

4.2 Requirements revisited

We can now revisit the requirements of applications demanding the management of multi non-independent quality of services in face of our definitions. The requirements are presented in Table 1.

Requirement RQ-1 is the basic motivation for our service. It is satisfied by allowing users to send messages over different channels. Requirement RQ-2 reflects the fact that, at the recipient site, the message should be processed by the peer entities of the protocols traversed during transmission. Requirement

RQ-3 imposes that inter-QoS constraints need to be taken into account and stacks with "diamond" structures need to be built. Finally, in requirement RQ-4, we state that micro-protocols should be combined without the need to re-write any code. This preserves the flexibility of the system, allowing an existent platform to be *extended* instead of *recreated*.

4.3 Defining stacks

A stack can be defined as a sequence of channels that may share sessions. Each channel is defined as a sequence of pairs (Layer,Session). Stacks can be easily defined in graphical form, such as illustrated in Figure 2. However, we believe that, for most practical purposes, graphs can be specified using a string with a simple syntax. We propose the syntax presented on Figure 3 for stack definition.

Channel is a name that identifies a given path, allowing system to satisfy requirements RQ-1 and RQ-2; *Layer* specifies a layer name and *Session* its instance. Session identifiers are simple integer numbers; note that the session identifier has a scope that is local to the stack specification since, by definition, channels that share a session of a given layer belong to the same stack. With this syntax users can:

- Use different instances of the same protocol for different channels by specifying different session numbers;
- Force channels to share layers by specifying the same session number.

Sometimes, multi-channel session may spontaneously generate events which are not bound *a priori* to a given channel. For instance, the intermedia synchronization layer may exchange non-audio and non-video control information with its peers. To avoid ambiguities, every multi-channel session must be associated to a *default* channel that is used to route all events which are not specifically bound to a given channel. In our example, control information could be routed by default through the data channel.

The stack graphically represented in Figure 2, would be specified as follows:

```
Stack = "Channel1:
                Layer[Session] |
                Layer[Session]
                ...;
        Channel2:
                ...;
        ...;
        Defaults:
                Layer[Session] -> channel;
                ...;
        "
```

Figure 3: Stack definition syntax

```
stack="audio:  APPLICATION[1] | INTERMEDIASYNC[1]
| AUDIOPROTO[1] | FIFO[1] | FAILDETECT[1] |
DEVICE[1]; video:  APPLICATION[1] |
INTERMEDIASYNC[1] | VIDEOPROTO[1] | FIFO[2] |
FAILDETECT[1] | DEVICE[1]; data:  APPLICATION[1]
| INTERMEDIASYNC[1] | DATAPROTO[1] | FIFO[3] |
FAILDETECT[1] | DEVICE[1]; defaults:
INTERMEDIASYNC[1] -> data"
```

Note that audio and video channels share sessions of the APPLICATION, MEDIASYNC and FAILDETECT layers but have their own local sessions of the FIFO layer.

4.4 Execution overhead

At first glance, one might think that the additional flexibility offered by our approach comes at the expense of some execution overhead. At this point, we still do not have a running prototype of our architecture. However, our experience with the Ensemble system lead us to expect a small overhead only during stack creation: the meta-structures required by our approach are more complex than those used in that system. On the other hand, the complexity of event routing should not be seriously affected.

4.5 Execution environment

The protocol stack can be configured to run both as a communications server or as a communication

library. The advantages of each mode can be enumerated:

i) The protocol stack will be executed in a separate process, with its own address space. This configuration can be found in several group communication frameworks, such as xAMp [4, 16]. It allows the same stack to be shared by several application on the same node but imposes additional context switching overhead.

ii) The protocol stack is executed in the address space of the application. Since the protocol stack needs to provide prompt response to network events, protocols threads should not be delayed by application threads. This approach works at its best when the system supports kernel threads.

The framework will be running outside the operating system kernel. This way, it will be possible to have applications running in different operating systems to interact.

4.6 Discussion

Previous work addressing the issue of preserving inter-stack constraints has solved particular problems. CCTL [15] focused on the membership problem. Maestro [3] has also given particular emphasis to membership and addressed some ordering issues, namely those related with message priorities. MMTS [5] incorporates the concept of " message bunches" that also allows to order data messages with regard to control messages. We take a more generic approach of allowing different protocol stacks to share an arbitrary number of layers.

Needless to say, our approach alone does not solve all the problems. The protocol (or application) designer still has to build the layers that perform the inter-stack synchronization. For instance, the "Intermedia sync" layer of Figure 2 has to be prepared to route events from different channels and synchronize them. Nevertheless, our approach allows complex inter-stack constraints to be addressed in an elegant manner through the composition of micro-protocols.

5 Conclusions

CSCW applications that support same-time/different-place interaction among users are bound to use several protocol stacks. However, these stacks are not independent. These applications often impose inter-stack constraints that can be implemented in an elegant way if the protocol stacks are allowed to share common layers.

In this paper we have described a communication architecture that allows the application designer to build the communication stack through the composition of micro-protocols in a way that allows him to express both inter-stack and intra-stack constraints. It is our belief that this architecture is particularly well adapted to the requirements of CSCW applications.

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high-availability. In *Digest of Papers, The 22nd International Symposium on Fault-Tolerant Computing Systems*, pages 76–84. IEEE, 1992.

[2] S. Barrett, B. Tangney, and V. Cahill. Constructing distributed groupware systems: A walk on the wilde side. Technical Report TCD-DSG#TCD-CS-1998-17, Trinity College Dublin. Distributed Systems Group, Sept. 1998.

[3] K. Birman, R. Friedman, and M. Hayden. The maestro group manager: A structuring tool for applications with multiple quality of service requirements. Technical report, Cornell University, Ithaca, USA, Feb. 1997.

[4] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM, Transactions on Computer Systems*, 5(1), Feb. 1987.

[5] G. Chockler, N. Huleihel, I. Keidar, and D. Dolev. Multimedia multicast transport service for groupware. In *Proceedings of the TINA Conference on the Convergence of Telecommunications and Distributed Computing Technologies*, pages 43–54, Sept. 1996.

[6] M. Correia and P. Pinto. Low-level multimedia synchronization algorithms on broadband networks. In *The Third ACM International Multimedia Conference and Exhibition (MULTIMEDIA '95)*, pages 423–434, San Francisco, Nov. 1995. ACM Press.

[7] F. Cosquer, P. Antunes, and P. Veríssimo. Enhancing dependability of cooperative applications in partitionable environments. In *Dependable Computing - EDCC-2*, Lecture Notes in Computer Science, pages 335–352. Springer-Verlag, Oct. 1996.

[8] S. M. W. Group. Synchronized multimedia integration language (SMIL) 1.0 specification. Technical report, World Wide Web Consortium, 1998.

[9] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University, Computer Science Department, 1998.

[10] N. Hutchinson and L. Peterson. Design of the x-Kernel. In *Proceedings of the SIGCOMM'88: Communications Architectures and Protocols*, Stanford, USA, Aug. 1988. ACM.

[11] W. Jia. Implementation of Reliable Multicast Protocol. *Software Practice and Experience*, 27(7), July 1997.

[12] S. McCanne and V. Jacobson. *vic*: A flexible framework for packet video. *ACM Multimedia*, pages 511–522, Nov. 1995.

[13] L. Moser, P. Melliar-Smith, A. Agarwal, R. Budhia, and C. Lingley-Ppadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, Apr. 1996.

[14] J. Postel. User Datagram Protocol. Technical Report RFC 768, USc Inf. S. Inst., Aug. 1980.

[15] I. Rhee, S. Cheung, P. Hutto, and V. Sunderam. Group communication support for distributed collaboration systems. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pages 43–50, Balitmore, Maryland, USA, May 1997. IEEE.

[16] L. Rodrigues and P. Veríssimo. xAMp: a Multi-primitive Group Communications Service. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 112–121, Houston, Texas, Oct. 1992. IEEE.

[17] B. Summers. *The Official NetMeeting Book*. Microsoft Press, 1999.

[18] R. van Renesse, K. P. Birman, R. Friedman, M. Hayden, and D. A. Karr. A framework for protocol composition in Horus. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 80–89, Ottawa, Ontario, Canada, 2–23 Aug. 1995.