DJINN: A Programming Framework for Distributed Multimedia Groupware Applications

Mitchell, Scott

ISSN 1369-1961

Department of Computer Science

Technical Report No. 727

DJINN: A Programming Framework for Distributed Multimedia Groupware Applications

Scott Mitchell

QUEEN MARY
AND WESTFIELD COLLEGE
UNIVERSITY OF LONDON

July 1996

DJINN: A Programming Framework for Distributed Multimedia Groupware Applications

by

Scott Mitchell, MCMS

First year PhD report

Distributed Systems Laboratory

Department of Computer Science

Queen Mary and Westfield College

University of London

July 1996

Abstract

In recent years multimedia has entered the world of computing to create a whole new class of application. The marriage of multimedia with groupware promises to bring exciting new capabilities to distributed cooperative working. These applications are on the leading edge of current technology, pushed ahead by advances in the performance and capabilities of computing hardware. Effective tools and methodologies for the specification, design and construction of multimedia applications are only now starting to emerge. Many applications are built in an ad hoc fashion, with resultant inflexibility and duplication of programming effort.

To diminish this problem, multimedia programmers can utilise a framework that encapsulates low-level details of the operating system and hardware, and provides a set of coherent and portable programming abstractions. As a step towards providing such a framework, this work proposes a comprehensive set of requirements for multimedia groupware programming. These are used as the basis for the design of DJINN, an object-oriented groupware programming framework. DJINN is intended to embody a bottom-up, exploratory approach to discovering appropriate high-level constructs for multimedia programming. Several demonstration conferencing applications have been built using DJINN, to evaluate its performance and limitations. The report concludes with a revised set of research goals, and a plan for future work related to DJINN.

Acknowledgements

I would like to acknowledge the support and contribution to this work of my supervisory committee, especially Professor George Coulouris. Credit is also due to Hani Naguib, without whom much of DJINN as it is today would not exist.

A special round of appreciation goes to June, for putting up with not seeing me for weeks on end. I only wish I could promise that it won't happen again! And finally, thanks to Tony Pennino for helping me over the rough patches and reminding me that there is much more to life than work.

Scott Mitchell

London, 22 July 1996

Table of Contents

List of Figures

1 Introduction

1.1. Motivation

Multimedia combines artifacts from different media (most often audio and video, but also including text, still images, animation and music) to enhance communication and enrich presentation [GIB95]. A multimedia presentation combines the various media elements using both spatial and temporal composition techniques. In recent years digital multimedia has emerged to create a whole new class of software applications, propelled by the availability of cheap audiovisual hardware on workstations and personal computers.

Groupware applications. also referred to as *co-operative* applications, are systems that allow multiple users to collaborate or interact. Typical examples of groupware are conferencing applications, shared drawing and editing programs, and group decision-support systems [COS96]. Most groupware research has fallen under the banner of CSCW (Computer Supported Cooperative Work). The growth in availability of powerful network- and multimedia-ready workstations means that distributed systems and multimedia technologies are finding an increasingly important role in groupware development. Groupware does *not* necessarily mean multimedia—the simple Internet 'talk' program is a groupware application—but its usefulness can be greatly enhanced by the addition of continuous media support

Multimedia applications lie on the cutting edge of current technology, driven by the rapid and continuous advances in the performance and capabilities of computing hardware. As with any new technology, the techniques to support the development and utilisation of multimedia applications are not yet fully understood [GIB95]. Multimedia applications—and multimedia groupware in particular—are characterised by being distributed, dynamic, real-time and interactive. These properties are generally not supported by the operating systems and programming environments in wide use today. Effective tools and methodologies for the specification, design and construction of multimedia applications are only now starting to emerge—not least because the space of applications is rapidly expanding. Many applications are built in an ad-hoc fashion, with resultant inflexibility and duplication of programming effort.

A programming framework or toolkit provides generic interfaces and mechanisms upon which specific applications environments can be implemented. Toolkits have already proved their usefulness in user interface programming, a notoriously difficult and time-consuming task. Multimedia and groupware applications share these, and other properties, so it is probable that well designed frameworks could also be beneficial in the area of multimedia programming. A framework encapsulates low-level details of the operating system and hardware, and provides a set of coherent and portable abstractions to application programmers. Applications using the generic mechanisms of the framework can adapt to changes in the underlying system, and new functionality can be added without disrupting existing applications.

1.2. Contribution

The research presented in this report is part of a larger project in the Distributed Systems Laboratory at Queen Mary and Westfield College. The overall goal of the project is to investigate system and programming support requirements for distributed groupware applications. To this end, the current research presents a preliminary design for DJINN, an object-oriented programming framework for groupware. Development of a prototype implementation of DJINN has commenced, and this implementation is evaluated by way of some simple groupware applications.

In particular, the design presented here includes support for multicast communications, and some experimental abstractions for high-level application structuring. A simple audio/video conferencing application has been built using the prototype implementation.

1.3. Overview of Report

The remainder of this report is structured as follows:

- **Chapter 2**, *Requirements*, examines in detail the requirements of a groupware programming framework. This leads to a set of research goals for this work and the larger project of which it is a part.
- **Chapter 3**, *Background*, presents a brief review of the literature directly relevant to this report. A companion survey document [MIT96c] explores the related literature in much greater depth.
- **Chapter 4**, *The DJINN Framework*, introduces the fundamental concepts of the DJINN groupware programming framework, and the reasoning behind this design.
- **Chapter 5**, *Implementation*, describes the work completed so far on the construction of the DJINN framework and the applications used to test it.

- **Chapter 6**, *Results and Evaluation*, presents the results of the framework testing and discusses these in the context of the research goals from Chapter 2.
- **Chapter 7**, *Conclusions*, summarises the contribution of this work so far, and proposes a plan for future research and development of the DJINN framework.

2 Requirements

This chapter presents a proposed collection of a requirements for a groupware programming framework. The emphasis is on providing support for real-time operation, quality of service management and portabilty rather than supporting a vast array of different devices and media. The concluding section of the chapter distils these requirements into a set of research goals for this and future work.

2.1. General Requirements

At a first glance, the requirements for a multimedia framework appear similar to those for any other programming framework. It must be simple and coherent, yet powerful and flexible enough to handle a wide range of tasks. Reliability and good documentation are also important considerations. There are however a number of multimedia- and groupware-specific requirements that must also be addressed. Gibbs and Tsichritzis [GIB95] present the following set of requirements for their multimedia programming framework:

- **Economy of concepts.** A multimedia framework should be based on a small number of concepts, otherwise there is the danger of it becoming a maze of media-specific details. It is particularly important to identify any general concepts that apply across media types.

- **Open.** It should be possible to extend a multimedia framework to incorporate new media type, new data representations, and new hardware capabilities as they become available.
- **Queryable.** A multimedia framework should specify interfaces for querying environments concerning their capabilities. Applications can then recognise missing functionality and adapt their behaviour.
- **Distribution.** A multimedia framework should help partition applications in a way that facilitates distribution. In particular, the objects within the framework should correspond to easy-to-distribute units or subsystems. Many future multimedia applications are likely to be distributed, so the utility of a framework is greatly diminished if it conflicts with distribution.
- **Scalable.** A multimedia framework should support scalable media representations. Once media are in digital form, improvements in quality and capabilities are tied to advances in hardware. To take an example, an early version of a processor might handle digital video at quarter-screen resolution and a few frames per second, while the next version is capable of full-screen resolution and 30 frames per second. If media representations are scalable, applications can increase quality as platform performance increases.
- **High-level interfaces.** A multimedia framework should provide high-level interfaces for media synchronisation, media composition, device control, database integration, and concurrent media processing activities. These operations are central to multimedia programming, so development is simplified if easy to use high-level interfaces are available.

The requirements listed above are quite general and are applicable to any multimedia programming system. However, the particular properties of groupware applications give rise to other more specific demands on a framework. To give an example, the interactive multi-party nature of groupware means that synchronisation mechanisms are needed to maintain consistency with respect to the actions of multiple distributed users. The following two sections present a set of requirements for a groupware programming framework and runtime support system for groupware applications. System support issues are addressed in this work in addition to the high-level programming abstractions, because it seems unlikely that one can succeed without the other. In the final analysis abstractions are only as useful as their concrete implementation in a real application, yet even the best runtime system will be under-utilised without a well-designed framework to simplify the programming task.

2.2. Framework Service Requirements

This section addresses the high-level requirements of a groupware framework, in terms of the services it presents to applications. A major concern of the framework is modelling various aspects of applications such as quality of service, data and control flow models, and user interaction. The framework must also provide mechanisms for dynamically building and configuring applications, and the essential services for their operation including communications, synchronisation and security.

2.2.1. Quality of service model

Applications have different communication and processing resource requirements. For example, a video stream in a conferencing application can tolerate more errors than a stream that is being recorded for future playback. Conversely, frames from the conference stream must be delivered on time, whereas this is not a concern for the recorded video [GIB95]. To support these differing requirements the framework should allow applications to specify their desired "Quality of Service" (QoS) parameters.

The high level model of QoS needs to know about all the available resources in the system, and how they are currently allocated amongst applications. It must provide a mechanism to convert application-level QoS specifications, such as "CD quality stereo audio" into the equivalent resource requirements for network bandwidth, processor cycles, etc. An application should be able to describe its desired structure and QoS parameters, and be told whether this configuration is possible or not. This "admission control" function can be performed before the application attempts any potentially expensive (both computationally and financially) allocation of resources that might not ultimately succeed.

Framework QoS services depend heavily on runtime support for resource management and admission control, discussed below. QoS issues are not specifically addressed in this research, but do form an important part of the broader DJINN project and are the subject of related ongoing research [NAG96, NAG96a].

2.2.2. Data and control flow models

Programmers make use of a groupware framework in two different ways. Primarily, application programmers will use the framework to build new applications from the existing components of the system. Less often, the framework itself will be extended to include support for new I/O devices, media types or networking technologies. In general application programmers are not concerned with the operational details of the components they use. However, these issues are important to the programmer building new components. The framework needs to specify a coherent policy (or set of policies) for the operation of its components to ensure that all parts of the system can interoperate. In particular, policies governing the transfer of control and data between system components must be well defined.

The buffering strategy to be used is an important part of the data flow model. Buffering is relevant to groupware applications because it has a significant impact on the latency, jitter and loss characteristics of communications. A buffering policy needs to specify *where* the buffering is to be done, and *who* is to provide the necessary buffer space. The actual transfer of data between system components is regulated by a flow control mechanism—for media streams *rate-based* schemes are usually superior to the *windowed* flow control used by protocols such as TCP/IP [POS81, STE94]. Whether to use sender- or reciever-initiated communications is another important question.

The control flow model specifies how many threads of control are to exist in the system, and how and when control is transferred between them. This work assumes that all machines support multiple threads of control within a single program, possibly with multiple processors. The overhead of thread switching can become excessive if too many threads are used, but on the other hand every opportunity for parallel execution should be exploited. Threading models are likely to be specific to the different platforms used in the system. It will nevertheless be useful for the framework to specify guidelines for component builders to encourage a consistent design approach.

2.2.3. Interaction model

Groupware applications by definition involve interaction with users, so consideration must be given to integration of the user interface with the multimedia aspects of the application. A programming framework should provide abstractions that allow for many different modes of interaction [GIB95]. Gibbs and Tsichritzis list a number of open research areas related to multimedia interfaces:

- **Multimodal interaction.** Multimedia applications should support the simultaneous use of multiple interaction channels.
- **Input streams versus input events.** Input processing in a multimedia system may be better handled using a stream-based approach rather than traditional discrete input "events".
- **Dialogue independence.** Applications should strive to insulate their internal model of interaction from the specific form of the user interface.
- **Interaction embedding.** Traditionally interactivity is a function of the application design. Hypermedia formats such as MHEG [JOS95] allow embedding of interaction directly in

multimedia data. This is suited to an object oriented approach where state and behaviour are naturally combined.

2.2.4. Application composition and configuration

An important property of multimedia—and especially groupware—applications is that they are *dynamic*. They need to be able to adapt to changing conditions in machines and networks, and to new or altered functionality. In addition, many groupware situations must allow for a changing number of participants, often within a single session. The framework abstractions must be as generic as possible to support this level of dynamic behaviour. Programmers should be able to concentrate on *what* their applications are doing, rather than *how* this is achieved.

Specifically this means that:

- Details of hardware devices, operating systems, network protocols and media formats should be hidden as much as is possible. An application should be able to request an audio and video link between machine *A* and machine *B* and have this set up regardless of where *A* and *B* are, and the specific hardware employed[1].
- When designing a complex program, the programmer will probably view the system from a much more abstract level than with a smaller application. The framework should allow applications to be composed at any level of detail, filling in the gaps itself where necessary. This would allow applications to be constructed recursively, using all or part of an existing system as a component of the new program.

Taking this notion a step further, the entire application could be composed using a high-level *specification language*, that would then be compiled into an appropriate implementation language.

[1]Within QoS constraints, of course.

Such a language would need to represent the dynamic configurability and quality of service properties of applications.

Various languages, such as Esterel, have been used to represent applications in terms of their synchronisation requirements. However none of these techniques adequately represents the dynamic, real-time nature of the applications and most do not deal with the distribution and communications aspects of a typical groupware system.

2.2.5. Group communications

Distributed, co-operating groups of users are a central concept of groupware, so effective mechanisms for group communications are essential. A framework should provide a multicast (one-to-many) communications service to augment the basic facilities of the underlying networks. Two important issues related to multicast communications are ***group management*** and ***message ordering*** [COU95, REN96]. Multicast groups determine who will receive particular messages, so clearly it is important that all members of a group have a consistent view of the group membership. Protocols exist for managing group membership, and for dealing with crashes and network partitions.

Message ordering specifies the relative order in which messages are delivered to a given group member. Most existing multicast systems offer ***unordered***, ***causal***, ***total*** and occasionally ***sync*** ordering semantics [COU95]. In general, message ordering is not an issue for media data, where delivering the data within QoS bounds is the most important thing. However, the situation is different for non-media data such as user interaction events or control messages [MIT96b]. While these data may still have real-time characteristics, they also affect the state of the application and thus will be subject to ordering constraints to ensure consistency.

The framework should include well-defined mechanisms for group management, and for sending and receiving multicast messages carrying media and non-media data. Other more complex communications services can be built on top of these. For instance, a video-conferencing application might use a M-to-M service in which M multicast groups share a small number of 1-to-M channels that distribute the images of currently active conference participants.

2.2.6. Synchronisation

Continuous media streams such as audio and video have both internal and external synchronisation requirements. Internal synchronisation is equivalent to *intra-stream* synchronisation, that is, maintaining the correspondence between the presentation of the stream data and the passage of real time [JAR95].

External synchronisation requirements can be divided into *inter-stream* and *event-based* synchronisation. Inter-stream synchronisation deals with maintaining temporal relationships between two or more related streams. These relationships may be natural—for example "lip-syncing" the audio and video tracks of a movie—or contrived, depending upon the content of the streams and their intended use. Thus requirements for inter-stream synchronisation are generally application-specific [AND91, SRE92]. An important aspect of inter-stream synchronisation is that the sources and/or sinks of the streams to be co-ordinated can be distributed across multiple sites.

Event-based synchronisation operates at a higher level, dealing with the co-ordination of activities described by stream- or user-related asynchronous *events*. Stream-related events include stream start/end indicators, frame timestamps and content-specific events such as a "silence marker" in an audio stream [SRE92]. User-related events cover user interactions with the application, for instance pausing playback of a video, or "floor control" in a conferencing application.

The problem of intra-stream synchronisation can be reduced to one of providing latency and jitter control over the entire end-to-end path of the stream. Here "end-to-end" is defined as including the source and sink devices as well as the network and any intermediate processing nodes. A number of techniques exist that, given suitable guarantees from the network, can provide effective intra-stream synchronisation [FER91, LIT91, ROT92, SRE92].

Inter-stream synchronisation aims to bound the temporal offset or *skew* between items from different streams that should be presented concurrently. In simple cases this can also be achieved by bounding latency and jitter on the streams in question. The situation becomes more complex when the sources and/or sinks to be co-ordinated are distributed over multiple machines. Most successful schemes [ROT92, SRE92, JAR95] use some form of adaptive rate adjustment to keep streams in synchrony. Control of rate adjustment may be effected by the source, using feedback information sent by the sink(s) [RAM93] or at the sink, often using a "logical clock" timestamping mechanism [AND91, ROT92]. Many techniques require at least approximately synchronised clocks at the source and sink.

While intra- and inter-stream synchronisation appear to be largely closed problems, the same cannot be said of event synchronisation. Event delivery can of course be bounded like any other real-time communication, but the semantics of the events themselves raise other questions. Ordered message delivery is certainly necessary, as discussed above. It is unclear whether events could be carried as stream data, or over their own independent channels, and whether they should be detected by source or sink nodes (or more likely, some combination of the two). Some research [SRE92, JAR95] has been undertaken in this area, but no clear conclusions have emerged so far. The semantics of events are very application-specific, but it seems clear that a service is required to detect, distribute and handle events while maintaining overall consistency of the application state. This is an important research issue for groupware and very relevant to the design of any programming framework.

2.2.7. Security

Security is a concern in any distributed system, especially one that crosses organisational boundaries as many groupware applications are likely to. While the algorithms and techniques for providing security are outside the scope of this work, it is advisable to consider how they might be integrated into the design of the framework. The main security related issues are *authentication*, *privacy* and *integrity* [COU94], all of which are relevant to groupware applications. A complete framework should offer at least the ability for users to authenticate themselves, and the option of encrypted communications where appropriate.

2.3. System Support Requirements

The purpose of the runtime support system is to implement the services of a framework on a particular multimedia platform. The framework abstractions are specialised for the hardware and software components provided by the target platform—although too much specialisation can lead to non-portable applications if certain features are not available elsewhere [GIB95]. Ideally much of the required runtime support would be provided by the operating system, but as discussed below this is unlikely in practice with most modern operating systems. The two main areas of concern for groupware applications are *real-time support* and the related but more wide-ranging issue of *quality-of-service* provision.

2.3.1. Real-time Support

Groupware applications are typically classified as "soft" real-time systems [KOP94]. They have temporal constraints on their behaviour, but it is usually acceptable for these constraints to be broken occasionally without permanent disruption to the programs operation. For instance, in most cases a user will not notice if a few frames of video are played back slightly late, or even lost entirely. The amount of tolerance allowable in these temporal constraints is both media and application specific. An application will require to level of service provided by the underlying system to be maintained within specified bounds. Guarantees made with respect to service provision must be honoured at all levels of the system:

Network support

A typical groupware application runs across a collection of geographically distributed sites, connected by a variety of local- and wide-area networking technologies. For reliable operation of the application the network needs to provide guarantees on the delivery of data to its destination(s). Specifically, the application expects to be able to deliver a certain quantity of data within a bounded time interval. The communications channel must supply sufficient bandwidth and low enough end-to-end latency to achieve this.

[illegible]

Some newer networking technologies—such as ATM—are able to make these guarantees, and techniques exist to bring a degree of real-time behaviour to normally non-real-time networks such as the Ethernet [MET76]. However, the TCP/IP protocol suite—and its biggest user, the global Internet—was not designed with real-time traffic in mind, and provides almost no support for real-time operation. Various attempts to rectify this situation are under way [VEN95, YAV94] but none has gained widespread acceptance to date. The lack of reliable real-time communication facilities is likely to remain for some time, adversely affecting any application not restricted to a single local-area or ATM network.

A network may offer other guarantees not strictly relevant to real-time transmission. Commonly these would include bounds on the number of bit errors or lost packets that might occur. It is also possible to bound delay jitter (the variation in transmission delay) by placing an additional, lower bound on latency, although this function is often provided by independent software components.

Guarantees such as these fall under the more general heading of Quality of Service (QoS) and will be discussed further below.

Operating system support

No matter how good the real-time facilities of the network, they are of little value if the application cannot keep its communication channels supplied with data, or fails to deal with incoming data in a timely fashion. Applications are engaged in processing as well as communication. and need assurances from the operating system that they will have enough time to carry out their computations. In modern multi-user operating systems the emphasis is on giving programs the illusion of full access to the machine, and sharing resources fairly amongst all users [BUL91]. This policy gives good performance on average, but on a heavily loaded system response times quickly degrade to unacceptable levels. This behaviour is unsuitable for multimedia applications, which often have legitimately "unfair" resource requirements, and need actions to occur at specific point in time.

The most important resources managed by any operating system are processor cycles and memory. Real-time applications require guaranteed access to both of these. To ensure that multimedia data are processed in time for their presentation to the user, a deterministic scheduling mechanism is necessary [VER93]. Each processing task will also use a certain amount of memory. Although guaranteed memory allocation is not strictly a real-time issue, having guaranteed access to it when required certainly is. If the necessary pages of virtual memory are not physically present, the applications CPU allocation might well have expired before the memory is paged in. Thus real-time applications require the ability to "wire down" virtual pages into physical memory to ensure timely access to their data.

Real-time operating systems do exist that provide these facilities. Examples include Chorus, Real-Time Mach, Lynx and QNX, all of which are derivative of the UNIX[2] system. The amount of

[2]UNIX is a trademark (at the time of writing) of X/Open.

real-time support offered by other UNIX variants and personal computer operating systems ranges from minimal to none.

2.3.2. Quality of Service Provision

The real-time considerations discussed above are but one aspect of the larger QoS equation. A complete QoS management system must also provide the following functionality in the runtime support layer:

Resource management

Each component of a distributed multimedia system offers certain resources to applications. The components and the resources they offer can be considered as a hierarchic structure, with the set of available machines and networks at the top level. Machines provide access to memory, processor cycles, network interfaces, media I/O devices and so on. Particular devices may specialise the offered resources still further. The resource management system is aware of:

- All the potentially available resources in the system, and where they are located.
- The quantity of resources reserved by each application.
- Upcoming "advance" reservations.
- Changes in the configuration of the underlying system, such as the addition or removal of machines.

This information is presented to the upper layers of the framework in a consistent and device independent format. The resource manager should also be responsible for making resource reservations on behalf of applications, although the applications themselves will need to actually allocate the reserved resources at the appropriate time.

Admission control

In a busy system there may be insufficient resources available to meet the current needs of an application. For example, a video-conferencing application might be able to allocate bandwidth for five audio streams but only two video streams. Ideally, an application would like to know if all of its resource requests will be successful before it begins to make any reservations. This functionality is usually provided by an admission control system [HER94]. An admission control algorithm takes a set of resource requests from an application and indicates whether or not these can be satisfied. More sophisticated systems might enter into a dialogue with the application, suggesting trade-offs that could be made between resources to allow the reservation to succeed. Once admission control has succeeded, the required resources should be reserved in one atomic operation.

2.4. Project Goals

As the previous sections show, the requirements for groupware programming and applications are quite extensive. Fortunately solutions do exist for many of these problems. However, a number of important research issues remain, in particular:

- **Application composition.** Groupware applications are often constructed in an ad hoc manner, using low-level, system-dependent techniques. There is a need for high-level abstractions that encapsulate specific functionality and allow applications to be build in a recursive and modular way. Ideally, such abstractions would be represented by way of a specification language from which working applications could be generated.
- **Synchronisation**, especially event based synchronisation. The exact nature of "events" in a groupware context is unclear, and generic mechanisms for synchronisation at this level do not yet exist.

- **Real-time group communications.** Most existing communications infrastructure provides no real-time guarantees for data delivery. It is uncertain how multicast communication, ordering and group membership semantics should be integrated with real-time transport protocols.
- **Quality of service support.** The broader issue of QoS provision is not well addressed by most modern operating systems. A full treatment requires real-time scheduling and I/O support, a resource management subsystem and mechanisms for QoS modelling and high-level specification.

The hypotheses of this research are the following:

1. *That a programming framework is an effective platform for evaluating ideas and building multi-media groupware applications, and*
2. *That a formal, high-level specification language for groupware applications is a useful tool for application design and quality of service modelling.*

Testing either of these hypotheses is necessarily a subjective task, because it is extremely difficult to quantitatively evaluate a set of programming abstractions or a specification language. Probably the best test that can be made is to define a set of applications (or classes of applications) that partition the domain of applications targeted by the framework or language. If it can be shown that each application (or an arbitrary application from each class) can be specified and implemented, then the language and framework can be deemed successful. Leaving aside the evaluation of the specification language for the moment, it may be possible to claim that a framework is "complete" if any application that can be expressed in the language can also be implemented in the framework.

The motivation for a notational language for application design is clear. Such a language would allow programmers—and maybe even non-programmers—to design applications at a very high level of abstraction without needing to be concerned with mundane and error-prone details of implementation. The relevance of the language to QoS modelling is perhaps less obvious. Quality of service becomes important as soon as the language starts to deal with issues such as synchronisation. It is not clear at this time how a model of QoS should relate to the actual QoS-controlled objects in a running application, and in any case that is not the subject of this research. Future work may determine that certain aspects of the QoS model are better expressed at a higher level than the framework itself can provide. In this case, these concepts should be integrated into the application specification language.

The work described in this report only specifically addresses the first of the hypotheses above. The second will be the subject of future research. Consequently, the goals of the current research are:

- Design and prototyping of an extensible object-oriented groupware programming framework (DJINN) upon which a variety of groupware applications can be built. DJINN is intended to allow an exploratory, bottom-up approach to the development of high-level programming constructs.
- Construction of one or more simple groupware applications to evaluate the success of the prototype framework.
- Specification of high-level abstractions for unicast and multicast communications, that meet the requirements of real-time multimedia applications. At this stage the formal specification of these abstractions is to take the form of a framework-level API (Application Programming Interface).
- Identification of multicast communications protocols suitable for real-time continuous media streams, and the integration of such protocols into the DJINN framework.

Chapter 7 reviews these goals in the light of work completed so far, and presents a plan for future research to meet these updated goals.

3 Background

3.1. Programming Frameworks

This section considers a number of systems providing support for multimedia or groupware programming. These range from the high-level group-oriented abstractions of GroupKit to the dedicated I/O devices and real-time communications of Pegasus and the Lancaster QoS architecture.

3.1.1. Gibbs & Tsichritzis

Gibbs and Tsichritzis' multimedia framework [GIB91, GIB92, GIB95] was developed at the University of Geneva. This framework will hereafter be referred to as the "G&T" framework. G&T is designed using an object-oriented methodology and implemented in C++. The framework classes are divided into four distinct groups:

1. **Media classes.** These correspond to the audio, video and other types of media supported by the framework.
2. **Transform classes.** These represent operations on media values.

3. **Format classes.** These deal with external representation of media data.

4. **Component classes.** These represent the devices and processes that modify media data.

G&T is designed to be portable and easy to extend. A great deal of effort has been put into defining a class hierarchy for static and temporal media types. The parallel hierarchy of format classes encapsulates the representation of media types on storage devices and over communications links. Transform classes represent operations that can be performed on media values, for instance image processing or video mixing effects. Transforms do not specify any timing constraints on the operations they encapsulate. Time-dependent operations are handled by the component classes. Components are *active* objects that apply temporally-constrained operations to media streams. Often, components will correspond to hardware I/O devices.

Applications are constructed in G&T by linking together networks of components using *connector* objects. Connectors provide communication between components, but do not offer any real-time guarantees. The issue of QoS provision in general is not addressed by the G&T framework—this is a significant limitation for real-time groupware applications.

3.1.2. GroupKit

GroupKit [ROS96] is a groupware programming toolkit from the University of Calgary, that allows developers to build synchronous and distributed conferencing applications. The design of GroupKit is based on the belief that groupware programming should be only slightly herder than programming similar single-user systems. GroupKit applications are written in Tcl-Tk [OUS94] and run on UNIX workstations.

GroupKit programmers are provided with three basic abstractions to support group working in their programs:

1. **Multicast RPCs.** These allow a group member to initiate an action at all or some of the application participants. Multicast RPCs are generally used to maintain state across the distributed application
2. **Events.** Events are used to notify conference participants when certain things happen, such as users entering or leaving the conference. Applications can define their own custom event types.
3. **Environments.** These are dictionary-style objects that can be used to provide a shared data structure within an application. Environments may contain ***active values*** that will be called when the information in the environment is modified.

These abstractions are mainly concerned with maintaining shared state between the distributed processes of a groupware conference. The toolkit also provides a selection of groupware widgets for building user interfaces, and session management tools. GroupKit does not have any support for continuous media, real-time operation or QoS control, so it is not directly relevant to the work described in this report. However, many GroupKit concepts are likely to be applicable to the higher-level design of DJINN.

3.1.3. Pegasus

The goal of the Pegasus project at the universities of Cambridge and Twente is to "...create the architecture for a general-purpose distributed multimedia system and build an operating system for it that supports multimedia applications." [MUL94]. Central to the system is the idea of a "multimedia workstation", consisting of a conventional workstation and various multimedia I/O devices connected to a local ATM switch. The switch is under the control of the workstation, and the ATM network forms the backbone of the entire system. Devices such as cameras and displays send and receive media data directly as ATM cells.

Multimedia workstations and processing nodes run a custom microkernel called *Nemesis*. This kernel provides support for multimedia applications in the form of timely scheduling and efficient interprocess communication. Nemesis is a 64-bit single address space OS, using an *activation-*based scheduling model [AND91]. Scheduling of user processes—called *domains*—uses a weighted discipline, with the weights calculated from user policies. EDF (Earliest Deadline First) scheduling is used to select between domains with remaining processor allocation. A QoS manager runs above the simple scheduler, with the task of updating scheduler weights to reflect changing application behaviour. An event mechanism is provided for inter-domain communications.

Pegasus implements interesting mechanisms for naming and invocation. The designers consider that naming conventions are more important than global name spaces, so the emphasis of the naming system is to provide efficient access to often accessed local names. Object invocations depend on the relationship between the invoking and invoked domains—simple procedure calls, 'protected' calls and RPCs are all available. Intelligent client stubs can, for example, cache values returned from remote objects, and object migration is supported.

While Pegasus does not provide any high-level support for multimedia programming, it introduces some useful ideas related to operating system support. Of particular interest is the use of dedicated multimedia I/O devices connected directly to a high-speed QoS controlled network.

3.1.4. Lancaster QoS Architecture

The University of Lancaster QoS-A (Quality of Service Architecture) is a "...layered architecture of services and mechanisms for QoS management and control of continuous media flows in multiservice networks." [CAM94]. The basic concept used in the QoS-A is the *flow*, which characterises the QoS-controlled production, transmission and consumption of a media stream. Flows can be unicast or multicast and carry both media and control data.

The QoS-A is divided into a collection of horizontal layers (distributed applications, orchestration [CAM92] and transport) and vertical planes (protocol plane, QoS maintenance plane and flow management plane). A QoS-enhanced transport service is provided, which allows users to negotiate a service contract with the network, using either deterministic, statistical or best-effort guarantees.

An ATM-based implementation of the QoS-A has been implemented on top of a modified Chorus microkernel, extended to provide support for QoS specification and real-time operation [COU95]. *Rtports* are standard Chorus communication ports with an associated QoS specification. User-supplied *rthandler* procedures are attach to rtports, and upcalled by the real-time infrastructure when data is received or requested. An advanced form of rthandlers, *QoS adaptation handlers*, allow applications to receive notification of QoS degradations or renegotiations.

3.2. Real-Time Communications

The systems summarised in this section provide support for real-time data transfer (both unicast and multicast), distributed resource reservation and multicast group management.

3.2.1. Tenet

The Tenet group, at the University of California at Berkeley and the International Computer Science Institute, has been studying multimedia communications since 1988. The research aims to design algorithms through which certain classes of networks can support real-time communications, then building protocols to realise these algorithms [FER92]. By 1992 the Tenet group had developed a scheme supporting real-time channels over hierarchic internetworks. A real-time channel is a simplex, unicast end-to-end connection with QoS guarantees. The scheme includes an algorithm for resource reservation and channel setup, using a traffic specification and set of QoS parameters supplied by the client.

In [FER92] the Tenet group outline a set of principles or beliefs on which their work is founded, and some essential features that follow from these. Briefly, the three underlying principles of Tenet are:

1. The interface offered by the network should be general.
2. The solution should be applicable to a wide variety of internetworking environments.
3. The clients of a real-time communication service require predictable performance.

These principles, and many of the protocol features derived from them, are applicable to other systems as well as Tenet. They should certainly be considered when developing detailed requirements for a real-time communication system for DJINN.

More recent work from the Tenet group has moved into other, high-level areas of groupware programming. For example [FER95] presents a system for advance reservation of resources in a real-time internetwork. This is built upon a later version of the Tenet protocol suite that provides *multicast* real-time channels as its basic abstraction.

3.2.2. ST2

The Internet Stream Protocol, Version 2 (ST2) is "...an experimental resource reservation protocol intended to provide end-to-end real-time guarantees over an internet. It allows applications to build multi-destination simplex data streams with a desired quality of service." [DEL95]. ST2 operates at the same level as IP [POS81] and allows bandwidth to be reserved along a network route. If network access and packet scheduling can also be guaranteed by hosts along the route, a well defined QoS can be provided. The ST protocols are similar to Tenet in that they follow a two-phase model: first resources are reserved using a non-real-time setup protocol, followed by the actual real-time data transfer.

ST2 is based around the concept of *streams*. These exist as simplex routing trees between a sending host and one or more receivers. Streams can be combined into groups to share allocated resources, or for failure processing. Stream construction can be sender-oriented, receiver-oriented, or a combination of the two.

Data transmission in ST2 is unreliable, on the assumption that sufficient resources have been allocated for a packet to reach its destination with the required QoS. The protocol makes no allowance for packets corrupted or lost by the underlying network.

3.2.3. RSVP

RSVP, the Resource ReSerVation Protocol [ZHA93], has some similarity to ST2, as it also endeavours to make resource reservations for multicast flows. Unlike ST2 however, this is all that RSVP does: it does not include any routing or data transfer protocols, nor restrict itself to any particular style of flow specification. RSVP is designed around six basic principles, listed below:

1. Receiver-Initiated Reservation.
2. Separation of Reservation from Packet Filtering.
3. Provision of Different Reservation Styles.
4. Maintenance of "Soft-State" in the Network.
5. Controlled Protocol Overhead.
6. Modularity.

One of RSVP's more unique properties is that reservations are made by receivers, rather than the sender as is usual in most other protocol. The arguments made for this approach are that firstly, the source can always transmit data, whether or not resources exist to deliver it. Receivers know the quantity of data they want or need to receive, and are after all the entities most interested in the QoS of incoming packets. Secondly, in a network where service is charged for, it is most likely the receiver who will be paying for whatever QoS has been requested. In this case the receiver would certainly like to be in control of what resources are reserved.

Reservations are made "backwards", in reverse along the path from sink to source. A reservation will only propagate back until it encounters an equal or greater reservation for data from the same source. In this way flows can be combined to conserve resources. Receivers may also apply 'filters' to their reservations. This allows, for example, use of reserved bandwidth to be restricted to packets originating from a particular participant of a conference—usually the person speaking at that moment.

RSVP is superior to ST2 in a number of ways. Firstly, RSVP takes advantage of multicast routing techniques that were not available at the time ST2 was designed. Because ST2 reserves resources along a tree routed at the source, using the same flow specification for the whole tree, it cannot easily accommodate receivers with differing QoS needs. Additionally, RSVP's filters provide a much more efficient solution to M-to-*N* reservation problems, such as conferencing.

3.2.4. RTP

RTP [SCH96] is a transport protocol for real-time applications, in one sense a companion protocol to RSVP, as it provides the data transfer functionality missing from the latter. RTP (Real-Time Transport Protocol) "...provides end-to-end network transport functions suitable for applications transmitting real-time data...over unicast or multicast network services." The RTP transport protocol is accompanied by a control protocol (RTCP) to allow monitoring of the real-time data delivery. RTP does not itself provide any delivery guarantees or any other QoS support. It is the responsibility of lower-level services to provide these features.

RTP supports unicast and multicast end-system to end-system communications, but more interestingly provides for RTP-level entities called *mixers* and *translators* to be inserted in the data stream reaching (some) recipients. A mixer might be used where some participants are only reachable over a low bandwidth link, to combine several incoming streams into a single, lower bandwidth encoding to be forwarded across the low-speed network. Translators are intended for applications such as 'tunnelling' through a firewall, and applications will usually not be aware of their presence. These entities have an obvious parallel in the 'transformer' components used by the G&T programming framework and, later, by DJINN.

The RTP control protocol (RTCP) provides regular updates on the performance of the transport system to participants. Each RTCP packet carries throughput, jitter and loss statistics for one or more senders or receivers.

RTP is very comprehensively specified in RFC 1889 [SCH96] and appears to offer a good solution to the transmission of real-time data over the existing Internet. Its usefulness is obviously limited by the lack of widely deployed protocols for resource reservation and admission control—although RSVP has potential to fill at least part of this gap. However, RTP is currently in use as the transport protocol for applications using the Internet Multicast Backbone (MBONE).

3.2.5. VDP

VDP (Video Datagram Protocol) is an RTP-based protocol developed at the University of Illinois, specialised for handling real-time audio and video over the World Wide Web [CHE96]. VDP is used by *Vosaic* (Video Mosaic), a real-time enhanced WWW browser that incorporates real-time audio and video into hypertext documents.

Within the context of the client-server model of the WWW, VDP aims to make efficient use of available network bandwidth and CPU resources at the client. The protocol takes advantage of the point-to-point connections between the web server and client. In this it differs from RTP, which allows multicast channels. Each VDP connections uses two channels: a high-bandwidth unreliable stream sending video data from the server to the client, and a low-bandwidth unreliable channel for feedback information from the client to the server. Data is sent from the server at its recorded frame rate (i.e. in real time). Frames are time-stamped by the server and buffered at the client.

Two algorithms are used to control delivery over the VDP connection. The first is a *rate adaption* algorithm that adjusts the transmission rate in response to changing network load, or insufficient CPU availability at the client. The client tracks the number of frames dropped because it could not process them quickly enough, and the number of packets lost due to network congestion. These statistics are reported back to the server which adjusts its transmission rate to reduce losses to an acceptable level.

The second algorithm is the *demand resend* algorithm which allows the client to request retransmission of certain frames, if these were dropped originally. For example, in a video using the MPEG coding scheme, the frames are grouped into sequences around the 'I-frames' which can be decoded independently of the rest of the stream. However, if an I-frame is lost a number of subsequent frames cannot be decoded, producing a gap in playback. The application may therefore choose to request that only I-frames should be resent. The server and client use quite large buffer queues, so it is likely that a resent frame can be inserted in the sequence before its playout deadline arrives.

It is unclear how well these algorithms would function in the multicast environment typical of groupware applications. In particular, the large buffers used by the demand resend system are inappropriate for 'live' streams requiring a low end-to-end latency. The rate adaption algorithm has possible application in a QoS degradation and renegotiation system.

4 The DJINN Framework

This chapter presents the fundamental concepts and abstractions of the DJINN programming framework, and the reasoning behind the design. The first section reviews Gibbs & Tsichritzis framework [GIB95], from which many of the basic ideas in DJINN are derived. The rest of the chapter examines in detail the DJINN abstractions that are relevant to this research. DJINN is not a solo effort—work related to QoS modelling and resource management is being undertaken in parallel with the research described here [NAG96, NAG96a]. A global view of the entire DJINN effort can be found in [NAG96b].

4.1. Overview

The main inspiration for DJINN is the multimedia framework presented by Simon Gibbs and Dionysios Tsichritzis in [GIB95], which was described in detail in the previous chapter. The media, transform and format classes of G&T allow the representation and transport of a wide variety of media types. However, the main attraction of the G&T framework for this work is the architecture of the component classes. This is based around the notions of *components* that encapsulate time-critical operation on media values, and *connectors* to transfer data between components. Components are *active* objects, that is, they are able to act spontaneously and thus introduce concurrency into the system. Each (useful) component provides one or more directional *ports* where data enters or leaves the component. Connectors are used to link pairs of ports.

The DJINN framework modifies G&T in a number of ways to provide support for:

- Quality of Service control
- Real-time operation
- Group-oriented communications
- High-level application composition and configuration

The following sections describe the major components of the DJINN architecture in greater detail.

4.2. Components

As with G&T, DJINN components are (potentially) active objects that generate and/or absorb media data. Some components can be likened to 'drivers' for hardware I/O devices, while others are purely software entities. Components encapsulate the time-critical operations specific to some media type. They are quality of service controlled objects.

Components use an event-driven control model. Currently two types of events are defined: data arrival at an input port, and signal or timer events. Signals can be generated by the arrival of a scheduling deadline, or in response to an external event such as a "frame ready" indication from a camera. Data transfer is producer-driven, to avoid the overhead of sending requests back along the data path—which may be costly or even impossible. Consumer-driven transfer is not specifically excluded, but the current design would make it expensive to implement.

4.2.1. Producers and Consumers

Producers and consumers generally correspond to hardware devices like cameras and speakers, although they may also be software-only objects such as a producer of animation frames. These components are also referred to as *sources* and *sinks*, respectively.

4.2.2. Transformers

Transformers are also known as *filters*. They both produce and consume media data, with the ouput(s) usually—but not always—some function of the input(s). Typical examples of transformers are a video mixer or a component that changes the frame rate of a stream.

4.3. Ports

Ports are the places where streams of media data enter and leave components. Ports are directional, and *typed* with respect to the kinds of media streams they carry. The purpose of ports is (obviously) to allow components to be connected together. Each port has an associated quality of service specification. This specifies the characteristics of the media stream *required* by an input port or generated by an output port, given the QoS parameters of the component as a whole. In general, port QoS specifications will only be changed by the component that owns the port.

DJINN ports share most of the criteria used by G&T to determine if a connection between two ports is possible:

- One port must be an input port and the other an output port.
- The media types of the ports must be compatible. This means that either the types are identical, or the type of the output port is a subclass of the input port type. This requirement is blurred somewhat by DJINN's use of high-level connectors that *can* change the media

type they carry. However, at the lowest level of application composition this constraint is maintained.

- The QoS of the two ports must be compatible. Currently "compatible" means that the QoS specifications of the ports are identical.

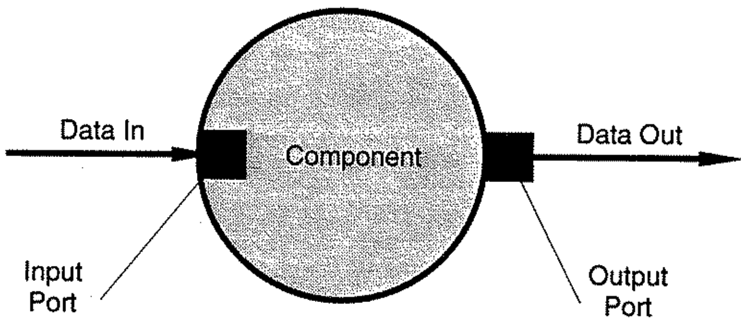

Figure 4.1. DJINN component with ports.

4.4. Connectors

In G&T, connectors are entities that provide a unidirectional link between a producing and a consuming component, via their ports. DJINN connectors still carry out this function but rather than being a separate class of object they are incorporated within the component hierarchy as a specialised form of transformer. There are several advantages to this approach:

- *QoS support.* Unlike components, G&T's connectors provided no support for QoS or real-time operation. By making connectors into components they gain some QoS support "for free" and ease the task of programming this functionality for specific type of connector.

- *Consistency*. Since connectors are components, they communicate with the outside world through ports. This means of course that applications are built by connecting pairs of ports rather than ports and connectors.
- *Permanent connections*. Because connectors now have "endpoints" (their ports) they can be created without reference to existing components as was previously required. This raises the possibility of setting up persistent connections that source and sink components can be attached to as necessary.
- *Connections without connectors?* Using this approach it is—in theory—possible to join two non-connector components directly, without an intervening connector. Alternatively, two connectors could be chained together. Whether either of these situations is actually useful is currently unclear, and a subject for future experiments.

Typical connector types are the *direct* connector that provides a direct, unbuffered link between two ports on the same machine, and the *UDP* connector which transfers data over a TCP/IP network using UDP datagrams. Connectors generally do not modify the data they transport, although some types may lose data due to network congestion or corruption.

4.4.1. Flows

Flows are an extension of the connector concept introduced by DJINN. A flow is a connector that encapsulates an arbitrary sequence of connected components. For instance, the connection between a camera and a display might be a flow made up of a video-compressor component, a network connector and a video decompressor, as shown in Figure 4.2. Any legal sequence of components can be encapsulated by a flow, so arbitrary transformations of media data can take place between its input and output ports.

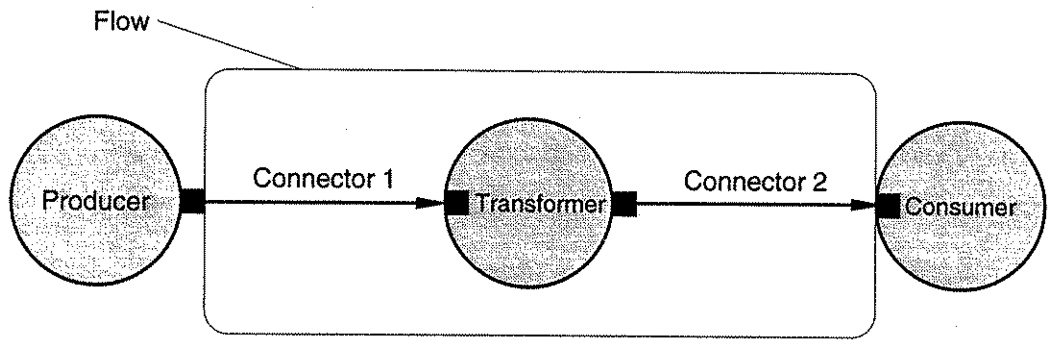

Figure 4.2. Example flow.

The rationale behind the introduction of flows is that they provide the programmer with a high-level structuring mechanism for building and manipulating applications. An application can treat a flow as either a distributed transformer component or a special kind of connector, without ever needing to worry about the details of the objects it contains. Requests and queries directed at the flow will be automatically passed on to the relevant encapsulated components.

The most important issue with flows is to determine exactly what components they encapsulate. Ideally, this decision would be made by DJINN at runtime when the flow is created. The runtime layer has up-to-date knowledge of available resources and is in the best position to know if the requested flow can be instantiated. How good or efficient a choice it makes is dependent on the underlying implementation. The application may know that it requires the use of some specific components, and it should therefore be possible to build a flow from a full or partial specification provided by the application. Another possibility yet to be considered in detail is for a flow to have multiple internal paths from source to sink. This might provide, for example, greater bandwidth by using multiple links, or improved performance by processing data in parallel.

4.4.2. Group Support: Transmissions

While flows allow a programmer to express application structure at a higher level of abstraction, they still permit only point-to-point connections between pairs of ports. G&T's ports allow the attachment of multiple connectors—this is not the case with DJINN, where ports can only be connected to a single other port. Multicast connectivity must be provided by the connectors themselves, and this is where ***transmissions*** come in. Transmissions are the multicast equivalent of flows: they provide a unidirectional connection between a single source port and *N* sink ports. Transmissions can be considered as bundles of flows sharing a common source and possibly some intermediate components. but they offer other more interesting capabilities:

- Transmissions co-operate closely with the multicast group management protocol. When a new member joins a group, a new sink port will be added to the corresponding transmission. The new connection must be validated by the admission control mechanism and have all its resources allocated before the group membership is updated.
- The QoS of an entire transmission can be changed with a single operation. The effect of this is to modify the ***maximum*** QoS that the transmission can deliver. Any sink that wishes to receive the transmission at a lower level of service is free to do so.
- Each recipient of a transmission effectively has access to a flow between itself and the transmission source. This flow can include arbitrary filter components, meaning that it is possible for every receiver of a transmission to see different media data.
- A transmission is just another type of connector, so it can be included as a component within a flow, or indeed within another transmission. This allows complex structures to be constructed easily yet still handled by applications as though they were simple components.

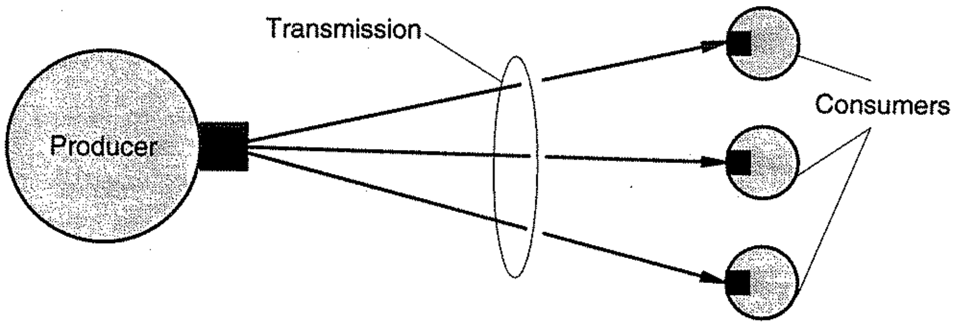

Figure 4.3. Example transmission.

The composition of connectors from other components can produce structures with ports other than the intended endpoints of the connection. Figure 4.4 illustrates how such a structure. might occur. The definitions of flows and transmission given above do not allow for this situation, providing no way to access these ports other than by examining the internal structure of the connector. This is a known shortcoming of the current design that will be addressed by future versions of DJINN.

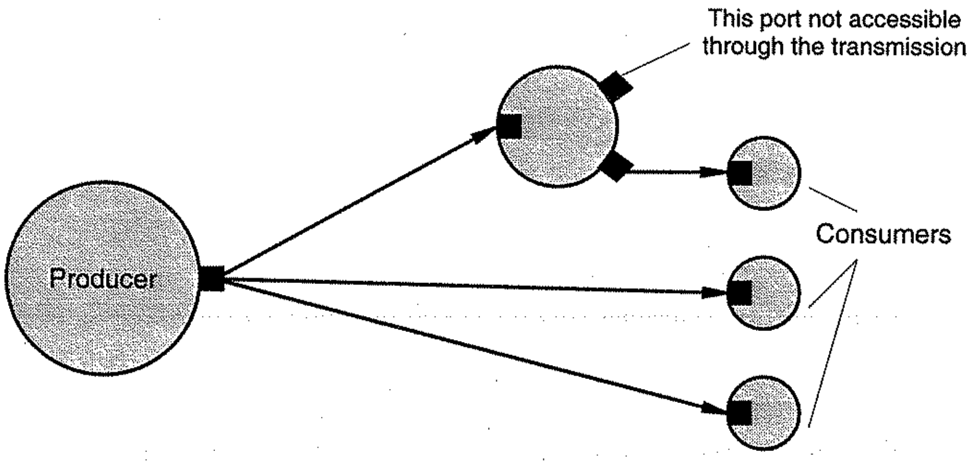

Figure 4.4. Complex transmission, with extra ports.

5 Implementation

This chapter describes the partial prototype implementation of DJINN that has been completed. The chapter begins with an overview of the environment and tools used to develop DJINN. Following this is a description of the "remote object" infrastructure used to provide distributed operation. The implementation of each of the major abstractions introduced in the previous chapter is summarised. The chapter concludes by presenting the groupware applications used to test the prototype.

Documentation for the current DJINN API can be found on the World Wide Web at the URL *http://www.dcs.qmw.ac.uk/~scott/Djinn/API/.*

5.1. Overview

The DJINN prototype is implemented in the Java[3] [FLA96, GOS96] programming language, on Silicon Graphics workstations running a UNIX-based operating system. The workstations are connected by a 10Mbps Ethernet LAN and are equipped with colour displays, video cameras, microphones and speakers. Java was chosen because:

[3]Java is a trademark of Sun Microsystems.

- It generates portable, architecture-neutral code. Although the code to access hardware devices is written in system-dependent C++, the compiled Java code will run unchanged on any Java-compatible platform.
- The language includes features such as multithreading and exceptions which make the programming task simpler and can produce a more efficient implementation.
- The standard class libraries include APIs for network access and graphical user interface creation.

The main drawback of Java is that—as an interpreted language—it is relatively slow, some 10–20 times slower than C++ [GOS96]. This is becoming less of an issue with the emergence of "just-in-time" compilers that should allow Java code to run at near-native speeds. In any case, time-critical code can be implemented in a compiled language if necessary.

The prototype includes experimental implementations of most major DJINN abstractions. Sufficient 'concrete' classes exist to build and run a simple multicast-aware conferencing application, described in the final section of this chapter. Figure 5.1 shows the hierarchy of classes that have been implemented so far (classes further down in the diagram inherit from those above). The figure omits a number of classes specific to the Java remote object system.

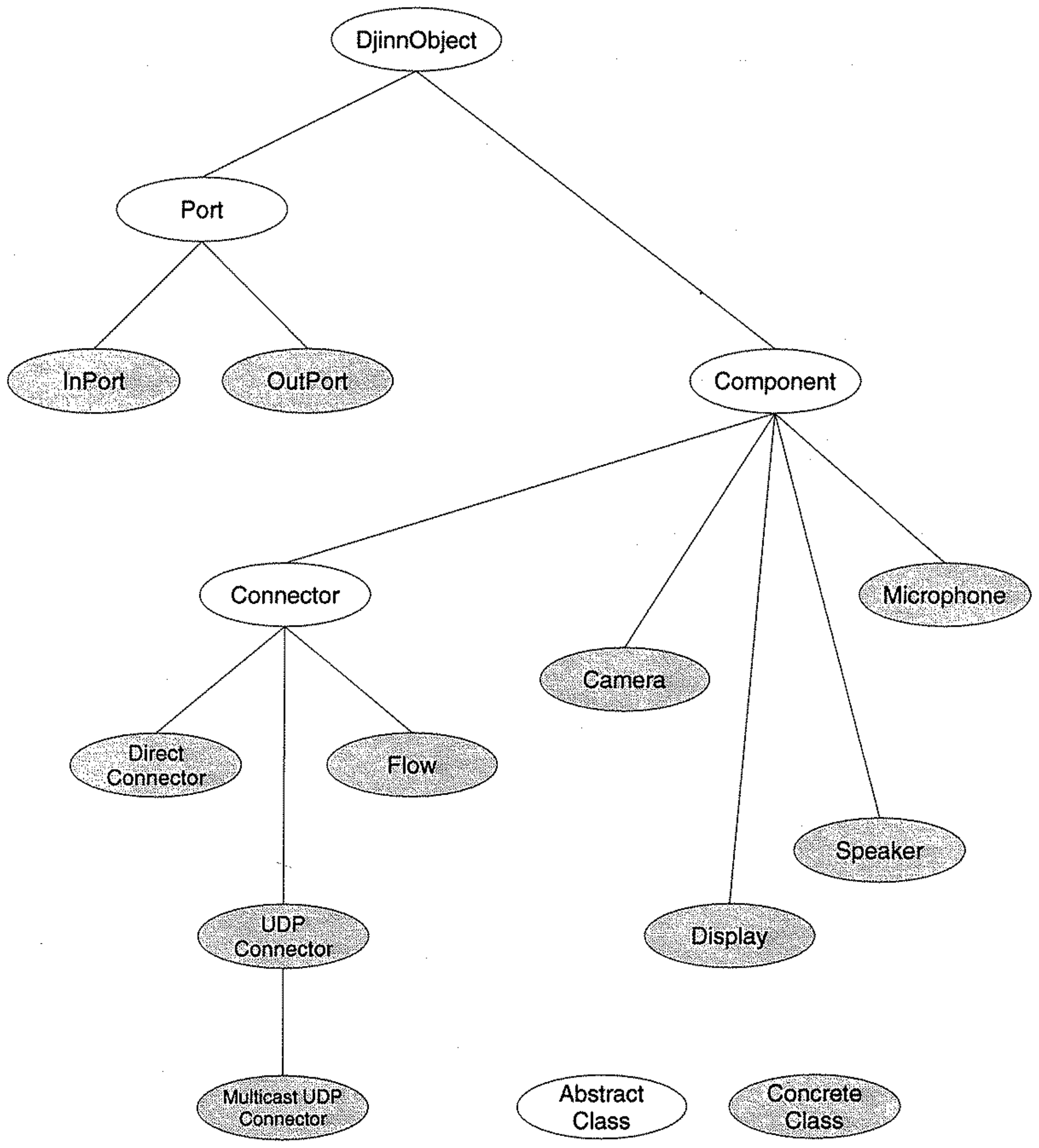

Figure 5.1. DJINN class hierarchy.

5.2. DJINN Remote Objects

DJINN uses Sun Microsystems' RMI (Remote Method Invocation) package to provide remotely callable objects—specifically objects that can be invoked across Java virtual machine boundaries. Remote objects are accessed though "remote references" that in most cases can be treated exactly like local objects. Most DJINN classes export methods that can be invoked remotely as well as local methods that can only be called by an object in the same virtual machine.

Every machine in a DJINN system must be provide a 'factory' object. This is an RMI object that creates new objects on its local virtual machine in response to requests from remote objects. The factory has a well-known name so it can be easily located on a given machine. Figure 5.2 below shows the relationship between local and remote DJINN objects and factories.

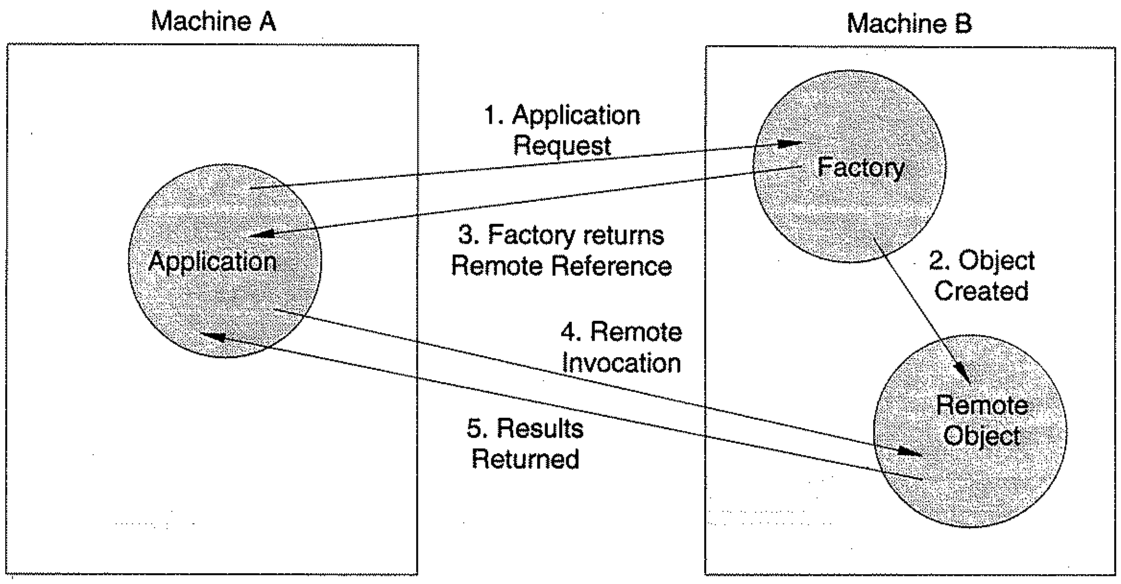

Figure 5.2. DJINN remote objects and factories.

5.3. Components

All DJINN components implement the following remotely invocable methods:

- **powerOn.** Prepares the component for use, allocating any necessary resources. The component assumes the resources have already been reserved on its behalf.
  Component-specific QoS parameters should be configured before calling **powerOn**.
- **powerOff.** This frees any allocated resources and shuts the component down. Most components can be successfully powered on again, perhaps with new QoS settings.
- **start.** Begins data transfer into, out of, or through the component, with the configured quality of service..
- **stop.** Stops data transfer. the **start** and **stop** methods can be called as many times as desired between a given **powerOn/powerOff** pair.
- **ports.** Returns a list of all the ports attached to the component.
- **findPort.** Looks up a port by name. Port names are specific to each class of component.

Components also implement local methods for data transfer and event handling.

A number of components driving actual hardware devices have been implemented. There are producer components for the Silicon Graphics camera and microphone. and corresponding consumer components for the display and speakers. These components augment the standard API with methods to configure device-specific QoS parameters, such as the camera frame rate or audio sample size.

5.4. Ports

In the current prototype ports act as little more than data forwarding agents. Output ports immediately transfer any data they are given to the input port they are connected to. Input ports simply notify their attached component that data has arrived. Clearly there are some inefficiencies in this setup, and this problem will be addressed in the next revision of the software.

Data is transferred in buffer objects, which are arbitrarily sized blocks of bytes. Buffer space can be allocated internally by a component or provided on request by a port. The buffering scheme has intentionally been kept very simple in this prototype, to avoid complicating the rest of the system.

Ports export the following remote methods:

- **connectedTo.** Returns a reference to the port that this port is connected to.
- **connectTo.** Connects this port to another port, which must be on the same machine. Currently no checks are made to determine if the two ports are compatible.
- **flowSpec.** Returns the "flow specification" of the port. The flowspec indicates the QoS of data transferred through the port. Currently flowspecs are expressed as a number of specific sized blocks of data per second. The flowspec is set by the component that owns the port.
- **locatedOn.** Returns a reference to the component that owns the port.

5.5. Connectors

Connectors inherit all of the basic component remote methods presented above, and add several new methods of their own:

- **init.** Limitations of the RMI system mean that no parameters can be provided at the creation of a remote object. This method is called after object creation to specify the two ports that the connector should link. Depending on the class of connector, this method may require the ports to be on particular machines, relative to the connector itself.

- **isConnected.** Returns a Boolean value indicating whether the connector represents a valid connection, that is, if data can currently be transferred through it.
- **source.** Returns a reference to the source (input) port of the connector. The reference could also be obtained using the component methods **ports** or **findPort.** This method is provided as a short-cut and may be removed from later versions of the API.
- **sink.** Returns a reference to the sink (output) port of the connector. As with the **source** call, this method is somewhat redundant.
- **flowSpec.** Sets the flowspec of the connector.

Two classes of connector have been implemented so far. The first of these is the *direct* connector. This connects two ports on the same machine as the connector. A direct connector provides no flow control or other QoS management—data is simply transferred immediately from the input to the output port.

A slightly more advanced connector exists in the form of the *UDP* connector. This provides an unreliable UDP datagram stream between two ports, which can be located anywhere with mutual IP connectivity. The UDP connector provides rate-based flow control if requested, sending and forwarding packets at the speed specified in its flowspec. UDP datagrams have an upper size limit of 64KB, so a simple packet fragmentation scheme has been implemented to deal with larger transfers. The connector currently makes no allowance for lost, delayed or out-of-order packets.

The UDP connector is an example of a *fragmented* object. A DJINN object exists at each end of the connection, and together these form the complete connector object. The state and behaviour of the connector are distributed between the two component objects. Both ends of the connector implement the same API, and will respond identically to requests.

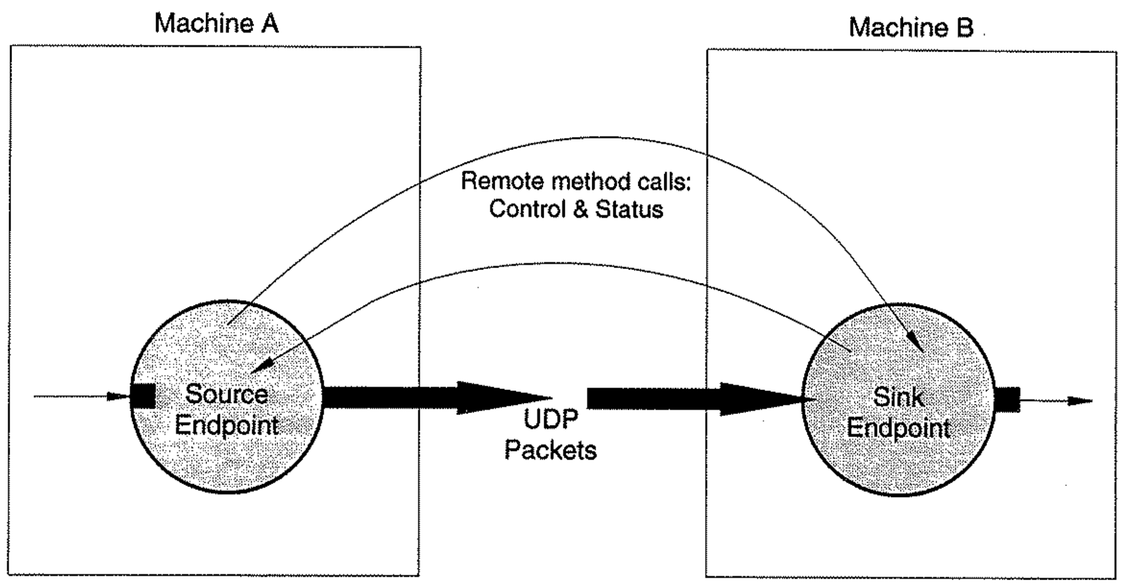

Figure 5.3. The UDP connector as a fragmented object.

5.5.1. Flows

At the application level, a flow can be used exactly like any other unicast connector. The recursive structure of the flow is only apparent when it is created. The prototype flow class provides several variants of the **init** method allowing the application to describe the internal structure of a new flow at different levels of detail:

- The original form of **init** introduced above specifies just the endpoints of the connection. The sequence of components to link the two ports is determined entirely by the flow.
- A list of components completely describing the path of the flow can be given to **init.** In effect, a 'wrapper' is placed around the listed components, allowing them to be treated as a single unit.

- The previous two forms of **init** can be combined to explicitly specify the connector endpoints as well as a set of internal components. The flow will create additional components as necessary to set up a link between the given ports. passing through the listed components.

Three additional querying methods, **allComponents**, **userComponents** and **systemComponents** return lists of components describing the entire flow, the user-specified portion and the system-generated portion, respectively.

The existing implementation of flows is very preliminary. It does not provide any support for atomically changing the QoS of a flow, or for flows with different input and output flowspecs. The generation of internal components is trivial—simple connectors are created to join the flow endpoints, and any user-specified components.

5.5.2. Multicast Connectors

Multicast capable connectors implement a number of new methods for managing their multiple sink endpoints:

- **init.** This is called after the connector object is created to specify the single *source* port of the connection. The two-argument version of **init** described above can also be used—this will configure the connector with a single initial sink.
- **sinks.** As the multicast equivalent of the **sink** method, this returns a list of all the sink endpoints of the connector.
- **numSinks.** Returns the number of currently connected sinks.
- **maxSinks.** Sets or returns the maximum allowed number of sinks for this connector. This value can be unspecified, in which case DJINN places no limit on the number of sinks.

- **addSink.** Attaches a given port as a new sink endpoint of the connector. The port receives any subsequent data sent through the connector. This operation will fail if adding another sink would exceed the maximum number allowed.
- **removeSink.** Removes the specified port from the connection.

DJINN does not currently define a group management protocol. Recipients can be added to and removed from multicast groups in an ad hoc way, without the other participants necessarily being aware of these changes. Additionally, no synchronisation or ordering constraints are provided. This means that a new recipient starts to receive data from some arbitrary point in the source stream, and that there is no way of determining when all participants have reached a particular point in the stream.

Changing the flowspec of a multicast connector modifies the flow produced at the source port. By default the flowspecs of all sink ports match that of the source, so that all sinks receive the same stream. More complex multicast connector subclasses may change this behaviour, allowing sinks to receive a transformed version of the original stream.

The prototype includes one concrete multicast connector class, the *MulticastUDPConnector*. This is an extension of the unicast UDP connector that uses the IP multicast protocol [DEE89] to deliver UDP datagrams to multiple destinations.

At the time of writing no support for transmission had been implemented.

5.6. Applications

Two versions of a simple audio/video conferencing applications have been implemented to test and evaluate the DJINN prototype. Both provide (by default) five frames per second of black-

and-white, 320 by 200 pixel video, and 16-bit single-channel audio at 8000 samples per second. The speed and quality of the audio and video can be adjusted if desired.

The first application serves as a demonstration of flows. It sets up a bidirectional audio and video connection between two workstations, using flows to make the connections. Figure 5.4 shows the structure of this application.

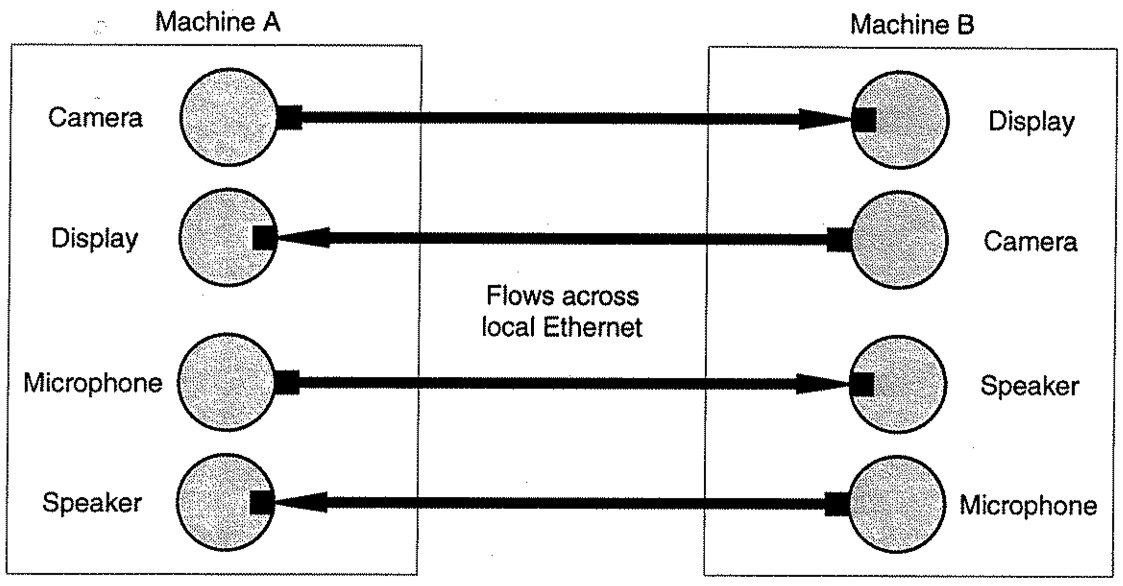

Figure 5.4. Simple conferencing application, using flows.

The second application is slightly more complex, and demonstrates the use of multicast connections. The program is split into two parts:

1. **A server process** that generates the outgoing audio and video streams on each participating workstation. The server creates a multicast UDP connector for each stream.
2. **A client program** to receive and present the data from selected servers. The client creates sink endpoints for the streams it is interested in and displays/plays these on the local workstation.

The normal mode of operation is for each participant to run both a server and client process on their workstation. The system also supports participants who run a client or server process only. A sample configuration of this application is shown below in Figure 5.5.

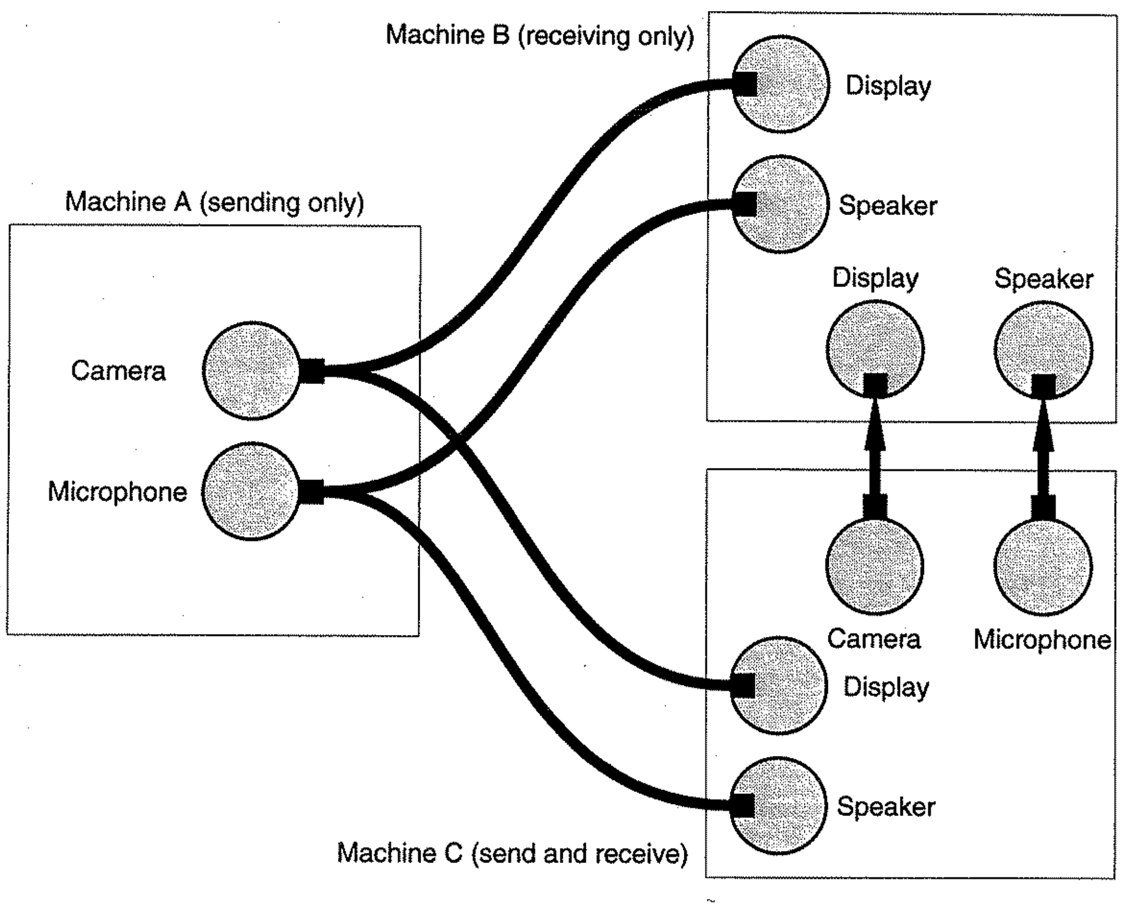

Figure 5.5. Multicast conferencing application.

6 Results and Evaluation

This chapter presents qualitative results from the framework testing described in the previous chapter. The DJINN design and the performance of the prototype implementation are evaluated from three different viewpoints: firstly, from the perspective of a user running a DJINN application; secondly, the view of the programmer writing the application; and finally, from the framework designers point of view, with respect to the requirements discussed in Chapter 2.

6.1. User Perspective

The demonstration applications described in Section 5.6 were designed to provide quarter-frame black-and-white video and telephone-quality audio. As implemented, the audio quality is better than required, using 16 bits per sample instead of eight.

Obviously, a conferencing application is useless if it cannot support multiple participants successfully. The multicast conference application performs adequately with up to three active participants—ie. three workstations sending audio and video streams onto the network. The number of recipients does not affect the performance of the application. With three sending machines, network collisions can cause video frames rates to drop, and excessive audio latency. With only two senders, the application performs to specification. The limit on performance is the 10Mbps bandwidth of the Ethernet connecting the participants. With three transmitting workstations the network is almost fully utilised[4].

Unfortunately the audio and video reproduction is not perfect. Audio playback will occasionally break up, and sometimes (partial) video frames are lost or arrive out of sequence. There are two main causes for these problems:

1. There is currently no resource reservation mechanism, so the application must compete with other programs on each workstation and other network users. If another program puts its packets on the network first, the conferencing system has no option except to wait, probably resulting in late arrival of the data. While the Silicon Graphics workstations have (in principle) more than enough CPU power to support this application, quite often another program will be scheduled on the processor at the time the conference needs to present or transmit some data, so again the data arrives late.

2. The Java runtime system is somewhat processor intensive, so that running two Java interpreters (for the server and client processes) simultaneously loads the machine quite heavily. This means that neither interpreter is receiving as much CPU time as it needs for optimum performance. Also, timing within a Java process is not particularly accurate. This leads to problems when attempting to do any kind of deterministic thread scheduling within Java. It may be that this is a result of the process losing the CPU at an inopportune moment, or simply the fault of the Java runtime system.

[4]Computing the required bandwidth from the figures above gives a result greater than 10Mbps. However, the video producer component tends to run slower than its nominal frame rate, due to inaccurate timing within Java. Thus the actual bandwidth is usually just less than 10Mbps, even allowing for UDP protocol overhead.

Clearly the performance of these applications could be improved by using a high-bandwidth network, or compressing the data before it is sent. Both of these possibilities should be investigated in the future. In a simple loopback experiment, with the camera and display on the same machine and no use of the network, the system was able to display video at 15 frames per second. At this speed the Java processes was consuming close to 100% of the available CPU cycles, so it is unlikely that a greater frame rate could be achieved without significant alterations to the runtime environment.

6.2. Programmer Perspective

From the point of view of the application programmer, using DJINN to build the conferencing applications was a relatively simple task. Both programs are only a few tens of lines in length, with most of these being methods calls to create, configure and activate components. It is trivial to extend and modify the behaviour of the applications.

Obviously a lot of non-essential features have been omitted from these simple demonstration programs. For example, there is no user interface (apart from the video windows), and any runtime errors will generally cause the entire program to crash. However, most of these issues are equally applicable to writing single user programs. Using a framework means that building a distributed groupware application is only slightly more complex than building a non-distributed application with similar functionality.

6.3. Requirements Evaluation

This prototype implementation of the DJINN framework did not set out to meet all of the requirements detailed in Chapter 2. In particular this research does not attempt to address any quality-of-service related issues, including synchronisation. It also does not deal with the user interaction model, security considerations, or the low level provision of real-time services.

Some progress has been made towards defining a data flow model for the framework. All communications are producer-driven, with buffering and jitter control generally handled at the sink end of connections. Rate-based flow control is available where necessary. The buffering model is not complete, as it does not specify which entities are responsible for allocating buffer space, nor the precise nature of the media data representation.

A model of control flow is less well developed, currently restricted to the use of Java threads to provide periodic events and background processing where required. The optimum configuration of threads and control flow is likely to be platform and operating system dependent, and is therefore not something that application programmers should be aware of. It is clear that more research and experimentation is required in this area.

Several useful abstractions for application composition have been proposed and implemented. Flows and transmissions allow programmers to deal with the 'what' of their applications, without needing to concern themselves so much with the 'how'. The simple implementation of flows appears to work well within the testing environment—the capabilities of flows can of course be extended and enhanced without affecting existing applications. It is unfortunate that pressures of time made it impossible to build a prototype implementation of transmissions.

The addition of multicast communications to DJINN was a relatively simple undertaking. Group communication will be further enhanced when transmissions are added to the framework. As yet, there is no support in DJINN for either multicast group management or message ordering. These issues are closely related to synchronisation, and should be addressed in that context by future work.

As regards Gibbs & Tsichritzis' general requirements for multimedia frameworks, most have been addressed by the current DJINN prototype. The framework has only a small number of basic concepts—components, ports, unicast and multicast connectors—and can easily be extended to incorporate new media and I/O devices. The requirement for queryability is only partially catered for. It is possible to query the configuration of individual components, but there is presently no way of, for example, finding the whereabouts of all video cameras in the system.

The framework inherently supports distribution, both through the use of RMI for remote object invocation and the provision of connectors that hide the details of networks and communications protocols. The scalability properties of the framework are uncertain. The current Java runtime environment would probably experience difficulty in providing a truly large scale distributed system, if only because of the large number of real time streams that might be needed. However. there should be no difficulty for applications to take advantage of increased platform performance as this becomes available.

DJINN provides a number of high-level interfaces to ease the task of programming complex multimedia applications. The most important amongst these are flow and connectors, which allow arbitrarily complex structures to be encapsulated and manipulated as single objects. The whole concept of components—and especially their arrangement into a hierarchy of abstract and concrete types—abstracts the details of device programming and lets programmers work at a much higher level.

7 Conclusions

7.1. Summary

In recent years multimedia has entered the world of computing to create a whole new class of application. The marriage of multimedia with groupware promises to bring exciting new capabilities to distributed cooperative working. These applications are on the leading edge of current technology, pushed ahead by advances in the performance and capabilities of computing hardware. Effective tools and methodologies for the specification, design and construction of multimedia applications are only now starting to emerge—not least because the space of applications is rapidly expanding. Many applications are built in an ad hoc fashion, with resultant inflexibility and duplication of programming effort. To diminish this problem, multimedia programmers can utilise a framework that encapsulates low-level details of the operating system and hardware, and provides a set of coherent and portable programming abstractions.

As a step towards providing such a framework, this work proposes a set of requirements for multimedia groupware programming. These are used as the basis for the design of DJINN, an object-oriented groupware programming framework. Several applications have been built using DJINN, to evaluate its performance and limitations.

7.2. Goals Achieved

The goals of this work, first stated in Chapter 2, were:

- Design and prototyping of an extensible object-oriented groupware programming framework (DJINN) upon which a variety of groupware applications can be built.
- Construction of one or more simple groupware applications to evaluate the success of the prototype framework.
- Specification of high-level abstractions for unicast and multicast communications, that meet the requirements of real-time multimedia applications, in the form of a framework level API.
- Identification of multicast communications protocols suitable for real-time continuous media streams, and the integration of such protocols into the DJINN framework.

While this research does not claim to have presented complete solutions to any of these goals, useful progress has been made towards the first three. Firstly, this report identifies a comprehensive set of requirements for multimedia groupware programming. Special emphasis was placed on requirements for QoS and real-time support, and high-level abstractions for communications and application structuring. These requirements provided the motivation for the design of DJINN, an object oriented framework for multimedia groupware programming.

Secondly, a prototype implementation of DJINN has been built, and used to construct several multimedia conferencing applications. The prototype is far from complete, and suffers from a number of limitations described in the previous chapter. However, it does allow programmers to build groupware applications at a relatively high level of abstraction, with only a little more effort than building a single-user application with similar functionality.

Thirdly, the DJINN design includes specifications of the flow and transmission constructs. These are high-level communications-oriented abstractions that allow complex structures (including entire applications) to be packaged and manipulated as a single DJINN object. The semantics of flows and transmissions have yet to be fully decided, but a provisional API for flows exists and has been implemented in the prototype system.

Regarding the fourth goal, a number of candidate protocols have been studied (see Chapter 3), but as yet no formal evaluation or comparison has taken place. This issue remains open and will be addressed in future research.

7.3. Plan for Future Work

A number of framework design and implementation issues remain. With respect to the current prototype version of DJINN, the following need to be completed:

- A general 'cleanup' of the code.
- Complete documentation of the API.
- A preliminary implementation of transmissions.
- More applications, ideally from a domain other than videoconferencing.
- Integration with Hani Naguib's work on QoS modelling [NAG96a].
- Quantitative evaluation of framework performance.

Once these tasks are completed, development of this prototype will be halted. The next version of DJINN should start afresh, acknowledging the lessons learnt from the development of the first prototype, but not necessarily using a great deal of its code.

Various framework requirements discussed in Chapter 2 have been ignored or only partly addressed so far. These include:

- Full specification of the semantics associated with flows, transmissions and other high-level application structuring and communications abstractions.
- The design of a group management protocol for multicast communications.
- The buffering and control flow (threading) models.
- Mechanisms for resource reservation and real-time service provision.
- The design of a query interface for the framework, to address the queryability requirement of G&T.

The final section of Chapter 2 presents two hypotheses for this research. The current work has only addressed the first of these. In order that solutions to both hypotheses can be properly considered, the following revised research goals are proposed:

- Continued development of the DJINN programming framework, as a basis for bottom-up research and experimentation with high-level programming constructs.
- Identification of multicast communications protocols suitable for real-time continuous media streams, and the integration of such protocols into the DJINN framework.
- Continued investigation of high-level constructs for group communications and application structuring. Some consideration should be given to the integration of these abstractions with mechanisms for QoS modelling.
- Investigate top-down approaches to multimedia groupware programming, including formal notations for specifying applications.

The approximate time-frame for this research is one year, i.e. from now until the end of July 1997. The work will be divided into the following phases (Figure 7.1 shows the distribution of work diagrammatically):

1. Complete the implementation of the current DJINN prototype. This includes documentation of the API.
2. Prepare a literature survey report, documenting the literature research carried out over the past year.
3. Co-author a paper related to the DJINN framework, to be submitted to a journal or conference.
4. Investigate current research in the areas of: abstractions for distributed systems composition and configuration; specification languages for distributed systems; protocols and constructs for group-oriented communications.
5. Design and implement high-level constructs for communications and application structuring in the DJINN framework. Preliminary specification of a notational language for groupware application design. This language may include aspects relevant to QoS modelling, dependent on the outcome of the related research to be conducted by Hani Naguib.
6. Testing and evaluation of the abstractions developed in the previous phase.

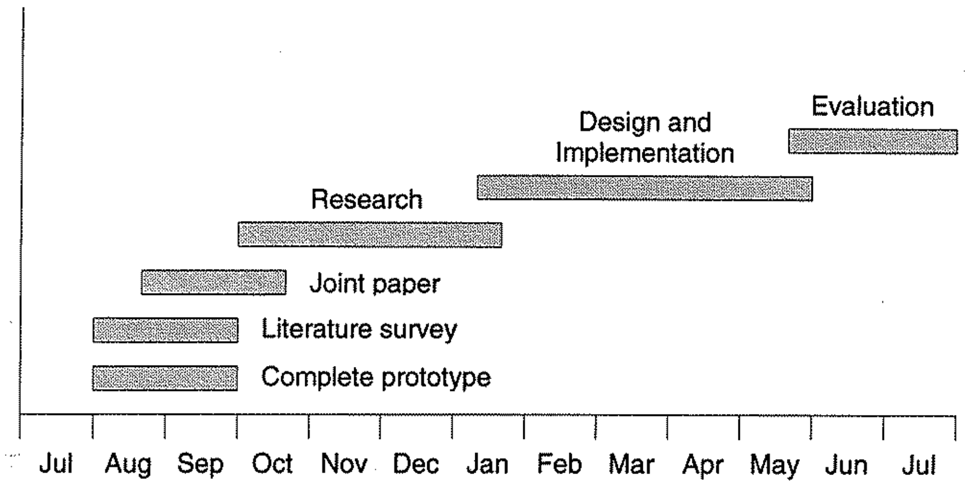

Figure 7.1. Proposed time-line.

References

[AND91] David P. Anderson & George Homsy. "A Continuous Media I/O Server and Its Synchronization Mechanism." *IEEE Computer*, October 1991, 51–57.

[AND91a] T. E. Anderson, B. N. Bershad, E. D. Lazowska & H. M. Levy. "Scheduler Activations: Effective Kernel Support for the User-Level Thread Management." *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, Pacific Grove, California. In ACM *Operating Systems Review* 25:5, October 1991.

[BUL91] Dick C. A. Bulterman. "Multimedia Synchronization and UNIX." In R. G. Herrtwich (ed), *Proceedings of the Second International Workshop on Network and Operating System Support for Digital Audio and Video*, Heidelberg, Germany, November 1991, 108–119. Lecture Notes in Computer Science no. 614, Springer-Verlag.

[CAM92] Andrew Campbell, Geoff Coulson, Francisco Garçia & David Hutchison. "A Continuous Media Transport and Orchestration Service." *Communications of the* ACM, August 1992, 99–110.

[CAM93] Andrew Campbell, Geoff Coulson, Francisco Garçia & David Hutchison. "Orchestration Services for Distributed Multimedia Synchronization." *IFIP Transactions on Computer & Communication Systems* 14, 1993, 153–168.

[CAM93] Andrew Campbell, et. al. "Integrated Quality of Service for Multimedia Communications." *Proceedings of IEEE INFOCOM '93*, San Francisco, California, March 1993.

[CAM94] Andwre Campbell, Geoff Coulson & David Hutchison. "A Quality of Service Architecture." Technical Report no. MPG-94-08, Department of Computing, Lancaster University, 1994.

[CHE96] Zhigang Chen, See-Mong Tan, Roy H. Campbell & Yongcheng Li. "Real Time Video and Audio in the World Wide Web." *http://choices.cs.uiuc.edu/New/vosaic/vosaic.html,* University of Illinois at Urbana-Champaign, 1996.

[COS96] François Jean Nicolas Cosquer. "Large-Scale Distribution Support for Cooperative Applications." Ph.D. thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa. March 1996.

[COU94] George Coulouris, Jean Dollimore & Tim Kindberg. *Distributed Systems: Concepts and Design.* Second edition, Addison-Wesley, Wokingham, England, 1994.

[COU95] Geoff Coulson & Gordon Blair. "Architectural Principles and Techniques for Distributed Multimedia Application Support in Operating Systems." *ACM Operating Systems Review* 29:4, October 1995, 17–24.

[DEE89] S. Deering. "Host Extensions for IP Multicasting." Internet Request for Comments no. 1112, Network Working Group, August 1989.

[DEL95] L. Delgrossi & L. Berger (eds). "Internet Stream Protocol Version 2 (ST2) Protocol Specification—Version ST2+." Internet Request for Comments no. 1819, ST2 Working Group, August 1995.

[FER91] Domenico Ferrari. "Design and Applications of a Delay Jitter Control Scheme for Packet Switching Internetworks." In R. G. Herrtwich (ed), *Proceedings of the Second International Workshop on Network and Operating System Support for Digital Audio and Video,* Heidelberg, Germany, November 1991, 72–83. Lecture Notes in Computer Science no. 614, Springer-Verlag.

[FER92] Domenico Ferrari, Anindo Banerjea & Hui Zhang. "Network Support For Multimedia: A Discussion of the Tenet Approach." Technical Report no. TR-92-072, Tenet Group, University of California at Berkeley and International Computer Science Institute, November 1992.

[FER95] Domenico Ferrari, Amit Gupta & Giorgio Ventre. "Distributed advance reservation of real-time connections." Technical Report no. TR-95-008, Tenet Group, University of California at Berkeley and International Computer Science Institute, March 1995.

[FLA96] David Flanagan. *Java in a Nutshell: A Desktop Quick Reference for Java Programmers.* O'Reilly & Associates, Inc., Sebastopol, California, 1996.

[GIB91] Simon Gibbs, et. al. "A Programming Environment for Multimedia Applications." In R. G. Herrtwich (ed), *Proceedings of the Second International Workshop on Network and Operating System Support for Digital Audio and Video,* Heidelberg, Germany, November 1991, 265–270. Lecture Notes in Computer Science no. 614, Springer-Verlag.

[GIB92] Simon Gibbs. "Application Construction and Component Design in an Object-Oriented Multimedia Framework." In P. Venkat Rangan (ed), *Proceedings of the Third International Workshop on Network and Operating System Support for Digital Audio and Video,* La Jolla, California, November 1992, 394–398. Lecture Notes in Computer Science no. 712, Springer-Verlag.

[GIB95] Simon J. Gibbs & Dionysios C. Tsichritzis. *Multimedia Programming: Objects, Frameworks and Environments.* Addison-Wesley, Wokingham, England, 1995.

[GOS96] James Gosling & Henry McGilton. "The Java Language Environment: A White Paper." *http://java.sun.com:80/doc/language_environment/*, Sun Microsystems Inc., Mountain View, California, May 1996.

[HER94] Ralf Guido Herrtwich. "Achieving Quality of Service for Multimedia Applications." *Proceedings of* ***, April 1994, 45–64.

[JAR95] Paul W. Jardetzky, Cormac J. Sreenan & Roger M. Needham. "Storage and synchronization for distributed continuous media." *Multimedia Systems* 3:4, September 1995, 151–161.

[KOP93] Hermann Kopetz & Paulo Veríssimo. "Real Time and Dependability Concepts." In Sape Mullender (ed), *Distributed Systems,* 2nd edition, Addison-Wesley, Reading, Massachusetts, 1993. Chapter 16, 411–446.

[JOS95] Robert Joseph & Jörgen Rosengren. "MHEG-5: An Overview." *http://www.fokus.gmd.de/ovma/mheg/rd1206.html*, December 1995.

[LIT91] Thomas D. C. Little & Arif Ghafoor. "Scheduling of Bandwidth-Constrained Multimedia Traffic." In R. G. Herrtwich (ed), *Proceedings of the Second International Workshop on Network and Operating System Support for Digital Audio and Video,* Heidelberg, Germany, November 1991, 120–131. Lecture Notes in Computer Science no. 614, Springer-Verlag.

[LIT92] Thomas D. C. Little & F. Kao. "An Intermedia Skew Control System for Multimedia Data Presentation." In P. Venkat Rangan (ed), *Proceedings of the Third International Workshop on Network and Operating System Support for Digital Audio and Video,* La Jolla, California, November 1992, 130–141. Lecture Notes in Computer Science no. 712, Springer-Verlag.

[OUS94] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, Massachusetts, 1994.

[MET76] Robert M. Metcalfe & David R. Boggs. "Ethernet: Distributed Packet Switching for Local Computer Networks." *Communications of the* ACM 19:7, July 1976, 395–404.

[MIT96] Scott Mitchell. "A Distributed, Quality-of-Service Controlled Network Service for Multimedia." Project proposal, Department of Computer Science, Queen Mary & Westfield College, February 1996.

[MIT96a] Scott Mitchell. "Supporting Group Communications in a Distributed Multimedia Groupware Framework." Project proposal, Department of Computer Science, Queen Mary & Westfield College, May 1996.

[MIT96b] Scott Mitchell. "Multimedia Stream and Event Synchronisation." Internal report, Department of Computer Science, Queen Mary & Westfield College, May 1996.

[MIT96c] Scott Mitchell (in preparation). "First Year Literature Survey." Internal report, Department of Computer Science, Queen Mary & Westfield College, August 1996.

[MUL94] Sape J. Mullender, Ian M. Leslie & Derek McAuley. "Operating System Support for Distributed Multimedia." *Proceedings of the 1994 Summer USENIX Conference,* Boston, Massachusetts, June 1994, 209–219.

[NAG96] Hani Naguib. "A Distributed Multimedia Framework." Project proposal, Department of Computer Science, Queen Mary & Westfield College, February 1996.

[NAG96a] Hani Naguib. "A Programming Framework for Distributed Multimedia Applications with QoS Support." Internal report, Department of Computer Science, Queen Mary & Westfield College, July 1996.

[NAG96b] Hani Naguib & Scott Mitchell (in preparation). "The DJINN Multimedia Groupware Programming Framework." Internal report, Department of Computer Science, Queen Mary & Westfield College, August 1996.

[POS81] Jon Postel (ed). "Internet Protocol DARPA Internet Program Protocol Specification." Internet Standard no. 5, September 1981.

[RAM93] S. Ramanathan & P. Venkat Rangan. "Feedback Techniques for Intra-Media Continuity and Inter-Media Synchronization in Distributed Multimedia Systems." *The Computer Journal* 36:1, 1993, 19–31.

[REN96] Robbert van Renesse, Kenneth P. Birman & Silvano Maffeis. "Horus: A Flexible Group Communications System." *Communications of the* ACM, April 1996.

[ROS96] Mark Roseman & Saul Greenberg. "Building Real Time Groupware with GroupKit, A Groupware Toolkit." ACM *Transactions on Computer Human Interaction* 3:1, March 1996, 66–106.

[ROT92] Kurt Rothermel & Gabriel Dermler. "Synchonization in Joint-Viewing Environments." In P. Venkat Rangan (ed), *Proceedings of the Third International Workshop on Network and Operating System Support for Digital Audio and Video,* La Jolla, California, November 1992, 106–118. Lecture Notes in Computer Science no. 712, Springer-Verlag.

[SCH96] H. Schulzrinne, S. Casner, R. Frederick & V. Jacobson. "RTP: A Transport Protocol for Real-Time Applications." Internet Request for Comments no. 1889, Audio-Video Transport Working Group, January 1996.

[SRE92] Cormac John Sreenan. "Synchronisation Services for Digital Continuous Media." Ph.D thesis, University of Cambridge, October 1992.

[STE94] W. Richard Stevens. *TCP/IP Illustrated Volume 1: The Protocols.* Addison-Wesley, Reading, Massachusetts, 1994.

[VER93] Paulo Veríssimo. "Real-Time Communication." In Sape Mullender (ed), *Distributed Systems,* Addison-Wesley, Reading, Massachusetts, 1993. Chapter 17, 447–490.

[YAV94] Rajendra Yavatkar, Prashant Pai & Raphael Finkel. "A Reservation-based CSMA Protocol for Integrated Manufacturing Networks." *IEEE Transactions on Systems, Man and Cybernetics* 24:8, August 1994, 1247–1258.

[ZHA93] Lixia Zhang, et. al. "RSVP: A New Resource ReSerVation Protocol." *IEEE Network Magazine,* September 1993.