Dynamically Reconfiguring Multimedia Components: A Model-based Approach

Scott Mitchell, Hani Naguib, George Coulouris and Tim Kindberg[1]
Department of Computer Science
Queen Mary and Westfield College
University of London

Distributed multimedia systems are potentially subject to frequent and ongoing evolution of application structures. In such systems it is often unacceptable for reconfigurations to fail or to only partially succeed. This paper describes the reconfiguration architecture of the DJINN multimedia programming framework. We introduce the concept of *multimedia transactions* for structuring changes into atomic units that preserve application consistency and extend these with the *smoothness condition* to maintain temporal as well as data integrity across reconfigurations. We present a technique for scheduling configuration changes that trades off the perceived level of smoothness against the available resources and the desired timeliness of the reconfiguration.

1. Introduction

The applications of distributed multimedia systems include complex and mission-critical domains such as digital television production, security and medical systems. These applications are potentially long-lived and are subject to frequent reconfiguration and long-term evolution of application structure. There is a clear requirement for support frameworks to manage the complexity of building and maintaining such systems.

This paper describes the DJINN multimedia programming framework [MNCK97], which is designed to support the construction and dynamic reconfiguration of distributed multimedia applications. DJINN applications have the novel property that they consist, seamlessly, of not only the components that perform media processing for the application itself, but also a model of these components. The model facilitates the construction of complex component structures; it allows us to perform QoS calculations conveniently; and it provides a vehicle for reconfiguring the application at run-time without breaking the application's integrity.

DJINN uses a split-level component architecture to model the structural and QoS configuration of distributed applications. High-level application structuring is based around hierarchic composition of components. Composite components use class-specific reconfiguration programs to modify the structural and QoS configurations of applications within constraints set by the component programmer. Configuration changes are performed on the application model, then passed through admission control and consistency checks before the changes are mapped onto the active peer component layer. This paper is primarily concerned with our techniques for achieving 'smoothness': maintaining the temporal consistency of media streams during reconfiguration by placing bounds on timing "glitches" that users may experience when an application is reconfigured. For example, consider a "tele-medicine" system where surgeons remotely conduct and observe a surgical procedure—it is clearly unacceptable for the video stream to break up when another observer joins the system. Smoothness is realised by imposing a schedule on component updates during reconfigurations, taking into account application- and system-imposed constraints on resource usage and trading these off against the desire to achieve a timely and glitch-free reconfiguration.

The remainder of this paper is organised as follows: Section 2 introduces the fundamental concepts of DJINN; Section 3 expands upon these and describes the basic mechanisms of reconfiguration in DJINN. Our algorithm for smooth reconfiguration is presented in Section 4 and discussed in Section 5. Section 6 briefly reviews related work and Section 7 contains our conclusions and plans for further research.

2. Building Distributed Multimedia Applications with DJINN

DJINN applications are constructed from networks of *active components* consuming, producing and transforming media data streams and interconnected via their *ports*, in a similar fashion to other distributed multimedia programming frameworks such as [Gib95, Bar96, FoSl96]. However, DJINN components come in two different flavours: *model* and *peer* components. Peer components are the familiar active objects that produce, process and consume media data streams. They are distributed according to the media-flow requirements of the application and generally have temporal constraints on their operation. Model components, on the other hand, are located wherever convenient for the application programmer or configuration management system. Their purpose is to model the configuration and QoS characteristics of the underlying peer components whilst hiding the details of

[1]{scott,hanin,george,timk}@dcs.qmw.ac.uk

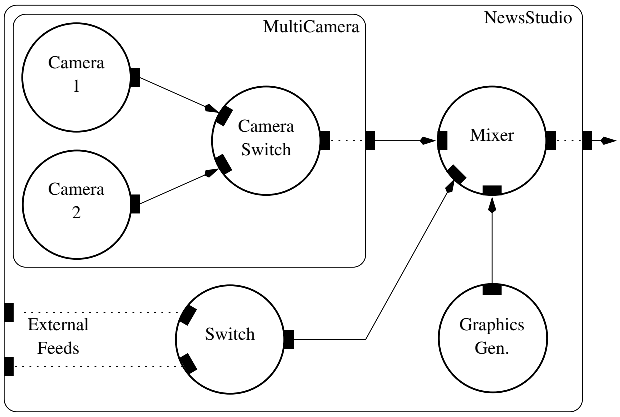

Figure 1. Model Components.

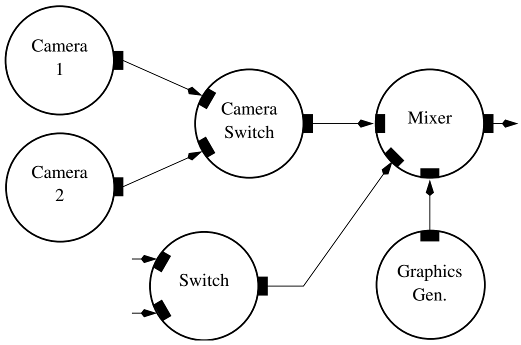

Figure 2. Peer Components.

media processing (transport protocols, media codecs, device drivers, etc.) from application programmers and users. Every peer component has a corresponding model component that knows about its configuration, QoS and media-handling behaviour. Additionally, *composite components* encapsulate sets of sub-components, for reasons of abstraction, modularity and reusability, and to provide high-level behaviours to applications.

It is important to understand that all application-level programming in DJINN takes place *at the model layer*; that is, application programmers interact only with the model components of an application. Peer components are created, configured and destroyed as required under the control of the application model. Figures 1 and 2 respectively show the model and peer components of an example "news studio" system that is part of a larger "TV studio" application. Note that while the peer components are distributed across several hosts according to the location of the necessary resources, the application model is located entirely on another host. Although the usual situation is for the complete model to be held on a single host our architecture does allow it to be distributed. This allows situations where parts of the system are controlled by different organisations; it also allows model components to be located close to their peers, for the sake of efficiency.

Our primary motivation for the use of an application model is to clearly separate the *design* of an application from its *realisation* at run-time [MNCK97]. The model is largely independent of location, hardware platform, operating system and the various technologies used to process and transport media data; it enables programmers to build and evolve applications at a high level of abstraction. Peer components, on the other hand, have no notion of their place in the larger application—they simply carry out their tasks of producing, processing, transporting and consuming multimedia data.

Components are controlled through a combination of remote invocations and inter-component events. Events can be transferred between components across the model-peer boundary and additionally may flow along the same paths as media streams, interleaved with media data elements.

3. Reconfiguring DJINN Applications

Composite components are the principal structuring mechanism for DJINN applications. These components encapsulate groups of sub-components that perform some higher level activity; this allows such groups to be replicated and instantiated elsewhere, even in a completely different hardware or network environment. The "MultiCamera" and "NewsStudio" components in Figure 1 are examples of composite components. As well as their encapsulation function, composite components can define their own class-specific behaviours on top of the basic "components and connections" model, permitting the creation of complex reconfigurations independent of the physical realisation of the model. Each composite component provides methods for managing its structural and QoS configuration, i.e. for manipulating the connectivity and per-component state of its subcomponent graph.

Application configuration—and reconfiguration—is expressed in terms of paths: end-to-end management constructs describing the media data flow between a pair of endpoints chosen by the application[2]. A path encapsulates an arbitrary sequence of ports and intervening components that carry its data; it also declares the end-to-end QoS properties of that sequence, including latency, jitter and error rate. It is up to each individual application to identify the end-to-end flows that are of interest to it and specify paths accordingly. Flows that are not part of a path do not receive any end-to-end guarantees either for their normal operation or during reconfiguration.

We have identified a set of requirements for configuring distributed multimedia systems, of which the following are the most important in the context of this paper:

[2]To represent 1-N configurations, a "bundle" of paths with a common source endpoint is used.

Atomicity. Often multiple related configuration changes must be carried out as a single operation, for example when adding a participant to a video conference. Atomic reconfiguration ensures that either all of the changes occur, or none of them do.

Consistency. Every reconfiguration should take the application from one *consistent state* to another. The properties of a consistent state are component- and application-specific—for example, the news studio system might specify that at least one input to the mixer must always be active.

3.1. Multimedia Transactions

The requirements for atomic and consistent reconfiguration have led us to the development of a transaction-oriented model of reconfiguration in DJINN. However, conventional database-oriented transaction mechanisms supporting the 'ACID' properties [HaRe83] have some shortcomings in the context of multimedia applications. Specifically, traditional transactions do not take account of the real-time, interactive nature of multimedia systems and the need to maintain temporal constraints on data—an important aspect of keeping the application in a consistent state.

Therefore, we introduce the notion of the *multimedia transaction* [Mit98] for managing the reconfiguration of multimedia applications. Multimedia transactions are an extension of the traditional transaction semantics. They continue to maintain the ACID properties, except for 'D' (durability) which is largely orthogonal to the aims of our current work[3]. To address the time-sensitive nature of multimedia, our transactions extend the usual notion of the 'C' property to include the temporal as well as data consistency of media streams. We have informally named this requirement to maintain temporal constraints across reconfiguration boundaries the *smoothness condition:*

> *"The execution of a multimedia transaction on a live system must not break any temporal constraint of any active path."*

Temporal constraints are application-dependent, but an application might specify, for example, that a new stream must have started to play out within 200ms of the old stream being stopped. We must be careful to distinguish between failures of temporal integrity and failures of data integrity. For instance, if the input of an MPEG decoder is switched to a new stream at the wrong time, a sequence of garbled frames may be output. This is obviously a failure of data integrity since the application has generated corrupt data. If, however, the badly-timed switch caused the decoder to pause for longer than an application-defined bound, then the smoothness property has been broken. Smoothness violations *may* also occur when an existing stream is stopped before its replacement has been started, or if two streams briefly run simultaneously when resources have been reserved for one stream only. Although the smoothness condition specifies that no temporal constraints should be broken, we believe that users will be prepared to accept a certain amount of degradation in service quality—equivalent to a temporary relaxation of constraints—around the time of a reconfiguration. For example, it is generally acceptable when switching between two video streams, for the interval between the last frame of the old stream and the first frame of the new to be longer than the inter-frame interval of either stream individually (that is, the jitter bound can be relaxed across the reconfiguration). The exact amount of degradation that can be tolerated—and for how long—will depend on the application and the preferences of the user.

Like all configuration-related activity in DJINN, transactions are carried out at the model component level, with their effects made visible at the peer layer when a transaction successfully commits. For a transaction to commit involves checking the consistency of the new configuration; admission control, with possible re-negotiation of QoS parameters; resource reservation; and updating of the affected peer components. Meeting the smoothness condition requires computing a schedule for the update operations on peer components; the derivation and execution of this schedule is described in detail in Section 4.

3.2. The Anatomy of a Reconfiguration

The actions needed to complete a reconfiguration can be quite time-consuming, especially if new components must be deployed and activated (this may involve setting up hardware devices as well as a considerable amount of remote invocation). However, in many reconfigurations there is a requirement that the transition between old and new configurations takes place either at an absolute real time or within some interval of someone "pushing the button"—both are difficult to achieve if there is an indeterminate amount of setup activity to perform before the switch can take place. Our solution to this problem is to divide the peer component updates performed by a multimedia transaction into two phases:

"Setup" phase. This phase is begins immediately when the transaction program is started. It encompasses all changes to the application model; structural and QoS checks on the new configuration; plus creation of new peer components and reservation of any resources they might require. The new peer components

[3] It would, however, be possible in principle to incorporate durability support into DJINN by using a persistent runtime support system such as PJama [Atk96].

are *not* connected into the running application and no existing components are stopped or deleted. Depending upon the smoothness requirements of the reconfiguration, some new components may be started running during the setup phase.

"Integrate" phase. This phase runs at some point after the setup phase has completed, in response to an event which may be generated at a particular time, by a user interacting with the application's UI, or by a peer component. The integrate phase completes the peer component update by "starting up" (instructing a component to begin processing, causing it to actually *use* the resources reserved for it) new components, connecting the new configuration into the running system and removing any components that are no longer needed, while ensuring that smoothness bounds are met. The event that activates the integrate phase of a transaction is specified when the transaction is started, along with a timeout value. If this timeout expires before the transaction has been integrated the effects of the transaction will be undone and the integrate event will be ignored if it ever does arrive.

This is obviously not a perfect solution, as applications still need to know approximately when reconfigurations will be occurring. In the fairly common case when this information is known, however, the technique greatly reduces the apparent time required to complete the reconfiguration.

Figure 3 below graphs an abstract "quality" metric for a single path against time, during a reconfiguration of that path. The value of the quality metric is an aggregation of the end-to-end properties of the path: latency, jitter and error rate. Obviously, in an actual reconfiguration the transitions between the different phases are unlikely to be as clearly defined. However, we believe that this graph is a good first approximation to the generalised behaviour of a path under reconfiguration.

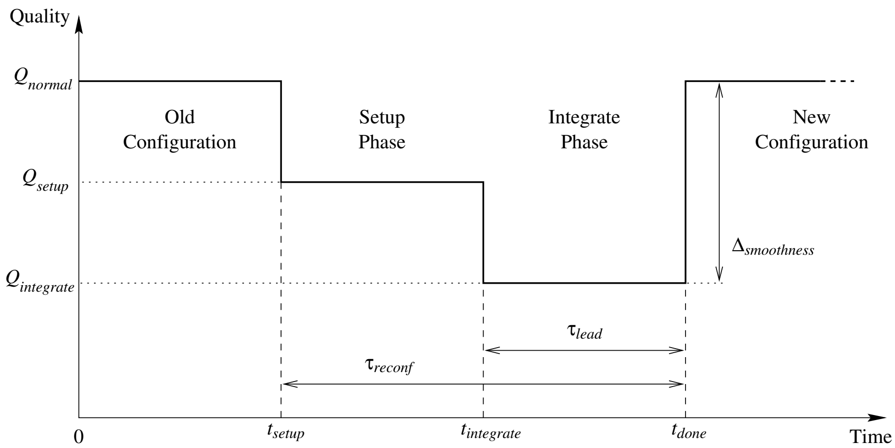

Figure 3. The anatomy of a reconfiguration.

We make several observations about the shape of the quality vs. time graph, which will lead us to a useful parameterisation of the reconfiguration in terms of smoothness, timeliness and maintenance of guaranteed service levels:

- The reduction in quality across the integrate phase of the reconfiguration, Q_{normal} - $Q_{integrate}$, determines the perceived smoothness of the transition between the old and new configurations. We define this quantity as $\Delta_{smoothness}$. Note that the service level delivered by the path is also potentially lowered during the setup phase, but the difference reaches a maximum during the integrate phase.
- The total time to complete the reconfiguration is given by $\tau_{reconf} = t_{done} - t_{setup}$. This corresponds to the total period of potentially reduced quality for the existing path. If the reconfiguration is being set up ahead of time, for integration at some point in the future, the setup phase may be of arbitrary duration.
- The elapsed time between requesting the integration of the reconfiguration and the path reaching its new final state (i.e., the length of the integrate phase) is given by $\tau_{lead} = t_{done} - t_{integrate}$.

For a given reconfiguration, a range of different schedules for the setup and integrate phases can be computed. These schedules trade off the three "reconfiguration quality" parameters τ_{reconf}, τ_{lead} and $\Delta_{smoothness}$, as well as resource availability, to determine the schedule that comes closest to meeting the applications' requirements with the least disruption to the rest of the system. In general, schedules providing 'better' smoothness (shorter lead time, or greater quality during the reconfiguration) will require more free resources to achieve. Given a particular set of available resources, there are also "internal" tradeoffs amongst the smoothness parameters; for example, a shorter lead time will almost invariably result in an increased value of $\Delta_{smoothness}$ over the integration period.

4. Achieving Smoothness

Consider the generalised two-switch-point reconfiguration shown in Figure 4, where an arbitrary path segment $I_k \ldots O_j$ in the middle of the path is replaced by another arbitrary path segment $I_k' \ldots O_j'$. The remainder of the path is completely unchanged by the reconfiguration. We will assume that none of the components in the *new* path segment exist before the reconfiguration begins. For this example we will define the path quality measure in terms of latency and jitter such that the latency between input port *i* and output port *j* or the path is given by $L_{ij} = \lambda_{ij} + \delta_{ij}$, where λ is the average latency and δ is the jitter. For a given pair of ports in either the old or new configuration, L_{ij} is the latency value guaranteed by the path (Q_{normal} in Figure 3) whereas L'_{ij} is the latency achieved whilst the *other* configuration is running simultaneously ($Q_{integrate}$ in Figure 3). We do not believe that it is feasible to calculate the effect over time of starting up some arbitrary set of components on the performance of the rest of the system. However, as a first approximation we can make the following two assumptions:

- In the case where the quantity of unallocated resources available is sufficient to meet the needs of the component being started, there will be no effect on already running components.
- In the case where there are insufficient unallocated resources available, we can determine a worst-case upper bound for the effect on already running components.

Therefore, L'_{ij} represents the *worst-case* latency on the path segment when the other configuration is running.

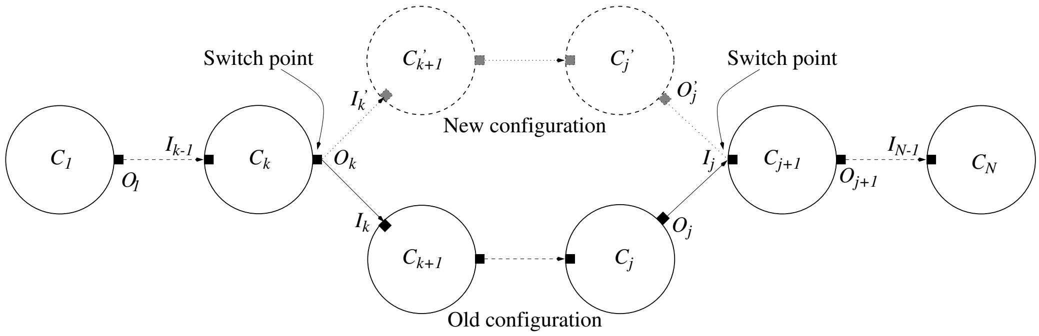

Figure 4. Generalised two-switch-point reconfiguration.

Components participating in the reconfiguration are 'primed' during the setup phase with a set of actions that each must perform during the integrate phase. Each component will carry out these actions upon receipt of an *integrate event*—either from an external source or on one of its input port—and optionally propagate the event downstream through one or more of its output ports. Integration is thus performed by scheduled delivery of integrate events to the farthest upstream points of the reconfiguration; the events will move ahead of the first new—or behind the last old—media data, triggering reconfiguration of downstream components as they go. The advantage of this approach is that we do not have to schedule the integration of each component individually, whilst still ensuring that the new configuration is integrated in time to deliver data to the rest of the application. In the reconfiguration of Figure 4, we have to schedule the starting of the new path segment, the stopping of the old, and switching to the new segment at O_k and I_j. O_k will be switched when the old path segment is stopped and I_j will be switched when the first new data arrives from O_j'. Thus, the schedule will consist of two events, e_1 (stop old segment) and e_2 (start new segment), delivered to O_k at times t_1 and t_2 respectively.

We now present a typical schedule for this reconfiguration, that attempts to strike a compromise between resource usage and improved smoothness and lead time. We assume that the new path segment is created at the start of the setup phase and that its post-integration resources reservations have succeeded.

4.1. A Typical Reconfiguration Schedule.

In this schedule we overlap the shutting down of the old path segment and the starting up of the new segment according to the difference between their latencies, as follows:

$$| (t_2 + L'_2) - (t_1 + L'_1) | \leq \Delta_{smoothness}$$

Whether e_1 or e_2 is delivered first will depend on whether L_1 is larger than L_2. In any case, the interval between the two events will be:

$$| t_1 - t_2 | = | L'_1 - L'_2 | \pm \Delta_{smoothness}$$

and the lead time will be:

$$t_{lead} = \max(L'_1, L'_2)$$

Note that we have used L'_1 and L'_2 in the equations above to reflect the fact that both path segments are running simultaneously for time $| t_1 - t_2 |$. L'_1 and L'_2 are worst-case values so we may in fact achieve better smoothness and lead time than is implied by the equations. The graph below shows the quality vs. time trace for this reconfiguration in the style of Figure 3.

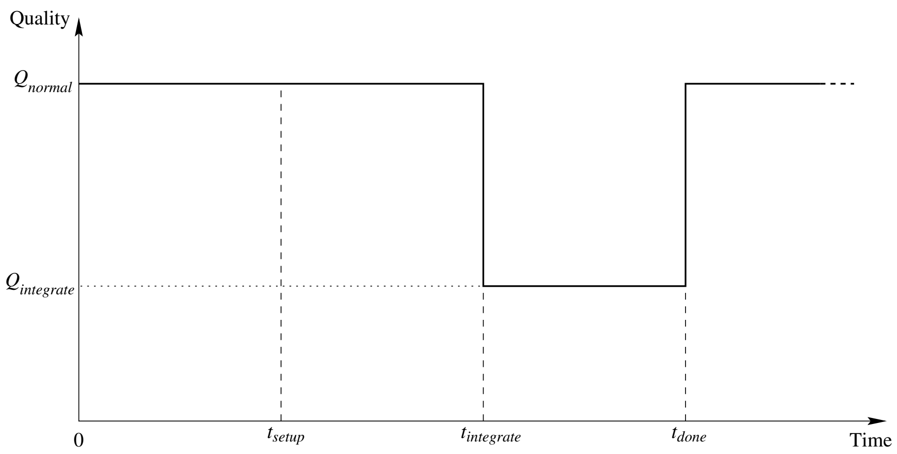

Figure 5. Quality vs. time graph for a typical reconfiguration schedule.

4.2. Alternative Schedules.

A variety of alternative schedules are possible. For instance, the TV studio or a security monitoring system probably requires near-instantaneous switching between camera inputs ($\tau_{lead} \approx 0$). This requirement can be met, at the expense of a large over-allocation of resources and potentially reduced smoothness if those resources are not available. To reduce the lead time, the new path segment is created *and started* during the setup phase. This means that the new data is being delivered almost to its destination even before the integration phase begins; the only action remaining is to switch from the old to the new path at port I_j. The schedule equations are similar to the previous schedule, except that the L'_2 term reduces to zero:

$$| t_2 - (t_1 + L'_1) | \leq \Delta_{smoothness}$$

The lead time in this case is effectively zero.

In an environment where resources are scarce and perhaps expensive—such as the Internet—we might wish to minimise resource usage by starting the new path segment only *after* the old segment has been completely shut down. This means that there is no simultaneous operation of the two configurations and thus no possibility of resource clashes. The disadvantage is, of course, that the reconfiguration will probably have very poor smoothness; all of t_{reconf}, t_{lead} and $\Delta_{smoothness}$ are likely to be large. The schedule is as follows:

$$t_2 = t_1 + L_1$$

That is, t_1 is unconstrained other than having to occur before the transaction timeout expires. The first new data will be delivered at:

$$t_{done} = t_1 + L_1 + L_2$$

The lead time will therefore be:

$$t_{lead} = L_1 + L_2$$

and the perceived smoothness at I_j will be:

$$\Delta_{smoothness} \approx L_1 + L_2, \text{ assuming that } \lambda_1 >> \delta_1 \text{ and } \lambda_2 >> \delta_2.$$

5. Discussion

The reconfiguration schedules outlined in the previous section deal only with the reconfiguration of a single path. Obviously, most real-world applications will consist of many interdependent paths and most reconfigurations will affect more than one of these paths. We believe that our techniques can be extended to work with multiple-path reconfigurations—this is a subject of ongoing research. The two most important issues to be considered in performing a multiple-path reconfiguration are:

1. Dependencies or constraints between the affected paths.
2. Sharing available resources amongst the paths during reconfiguration.

A common inter-path dependency is synchronisation, such as lip-syncing of audio and video paths. We assume that a conventional end-to-end jitter control and inter-stream synchronisation algorithm [Refs.] is in operation. We must ensure that the reconfiguration does not allow the paths to drift out of sync with each other by an amount greater than the synchronisation mechanism can handle. This implies scheduling the changes to each path so that the reconfigurations complete at the same time. For instance, if path P_1 has $t_{lead} = x$ and path P_2 has $t_{lead} = y < x$ we would inject the integrate event for P_1 at time t and that for P_2 at time $t + (x - y)$. This calculation can be extended to an arbitrary number of paths needing to be synchronised.

When resources are scarce the effects of a multiple-path reconfiguration on existing media flows are likely to be even more severe than in the single-path case described above. Given that there is a limited quantity of resources available for maintaining smoothness, we must decide how to distribute these amongst the paths that are being reconfigured. This is a decision that should ideally be made by the individual application, since only it is in a position to say which paths are more "important" than others and should be given a higher priority in terms of resource allocation. For instance, in a conferencing application it is more important to have glitch-free audio switching at the expense of a few dropped video frames; the opposite may be true for a security camera monitoring system. Therefore, in addition to specifying their desired smoothness and lead time for a reconfiguration, applications should be able to specify a "resource priority" for each of the paths involved. Further experimentation is necessary to determine whether this prioritisation should be performed on a per-resource basis or globally per path and to develop an appropriate algorithm relating priority to eventual resource allocation.

A potential difficulty that was ignored in the previous section arises from the fact that some media data streams cannot be switched at completely arbitrary points. The frames of an MPEG video stream, for example, are only meaningful in the context of the preceding *I*-frame; thus such a stream should only be switched to a new destination if an *I*-frame is the next element to be produced [BBH97]. This obviously affects our scheduling equations, since a path's response to the receipt of an integration event may have to be delayed until an appropriate point in the media stream is reached. The effect will be an increased lead time for the reconfiguration and a possible loss of synchronisation if more than one stream is involved.

Our examples so far have dealt with live rather than pre-recorded media streams. In general, achieving glitch-free reconfiguration for live streams is often considerably more difficult, because the length of the stream is not known in advance or because the latency requirements for an interactive application limit the amount of buffering that can be used to "smooth over" potential glitches. If, however, we are switching between stored streams we can start the new configuration running during the setup phase, at a reduced rate—requiring very few resources—and draining into a buffer close to its eventual destination. At integration time, the new stream can start playing *immediately* from the buffer, while we reconfigure according to the minimum resource usage schedule described earlier. The only constraint is that the buffer hold sufficient data to completely mask the lead time of the reconfiguration. Similar optimisations are possible in most situations utilising stored streams.

6. Related Work

The component-based approach to application construction is used by a variety of multimedia programming frameworks, such as G&T [Gib95], Medusa [WGH94] and CINEMA [Hel94]. CINEMA is in many ways quite similar to DJINN, as it makes use of composite components and a separate 'model' of the application—quite separate from the media handling components—that is used for control and reconfiguration [Bar96]. CINEMA's idea of what constitutes a reconfiguration is quite limited and certainly not transactional. It has no equivalent of the 'smoothness' property for ensuring clean transitions between consistent states. The need for smoothness support in the real-world domain of digital television production—where there is a requirement to "splice" together MPEG streams within the resource constraints of hardware decoders whilst still meeting QoS guarantees— is illustrated by [BBH97].

In [SKB98], Sztipanovitz, Karsai and Bapty present a similar two-level approach to component-based application composition in the context of a signal-processing system whose applications share many of the real-time requirements of multimedia. Their system performs adaptive reconfiguration by having several alternative configurations embedded in the model, which is also associated with a finite state machine whose states are assigned to the different structural alternatives.

Reconfigurable distributed systems have been an active area of research for more than a decade, with the Distributed Systems Engineering group at Imperial College London at the forefront of this field. Work produced by members of this group [KrMa90, FoSl96] has tackled many of the issues relevant to DJINN: consistency maintenance via transactional reconfiguration; dynamic reconfiguration; configuration of large-scale, long-lived systems; and the tradeoffs between programmed and interactive reconfiguration. However, the DSE group's research has primarily dealt with general-purpose distributed applications and does not explicitly address the real-time requirements of multimedia systems. Non real-time systems can afford to temporarily shut down the parts of an application undergoing reconfiguration—by moving them to a *quiescent* or passive state [GoCr96, Kin93]—without breaking any integrity constraints. Central to the Imperial College work is the notion of a *configuration language* for specifying the allowable configurations and reconfigurations for an application. Configuration languages are also an important aspect of other research, such as that conducted by the SIRAC project [BeRi95, BABR96].

7. Conclusions and Future Research

This paper has concentrated on the calculation of schedules for updating peer components in order to achieve a glitch-free and timely reconfiguration of multimedia applications within the resource constraints of the underlying system. Our main contribution is the addition to the DJINN framework of multimedia transactions, which take applications from one configuration to another while maintaining their temporal consistency as well as their structural and data consistency. DJINN's use of model components is crucial to the success of multimedia transactions—beyond its existing benefits [MNCK97] a model allows the system to validate proposed reconfigurations without affecting running components and to integrate changes in a principled way that trades off QoS guarantees, resource availability and time constraints according to application preferences.

We have implemented the DJINN model framework in Java, and are targeting a real-time distributed system running the Chorus kernel and supporting peer components written in C++. We are currently engaged in instrumenting the peers to track their resource usage and real-time performance as a means of validating our approaches to QoS management and reconfiguration. Future research planned with respect to smoothness includes extending and validating the approach described here for multiple paths with inter-path constraints, and further exploration of the tradeoffs between resource usage and reconfiguration quality in larger, real-world applications.

References

[Atk96] M.P. Atkinson, L. Daynès, M.J. Jordan, T. Printezis & S. Spence. "An Orthogonally Persistent Java." *ACM SIGMOD Record* 25(4), December 1996.

[BABR96] Luc Bellissard, Slim Ben Attallah, Fabienne Boyer & Michel Riveill. "Distributed Application Configuration." *Proc. 16th International Conference on Distributed Computing Systems*, Hong Kong, pp 579–585, May 1996.

[Bar96] Ingo Barth. "Configuring Distributed Multimedia Applications Using CINEMA." *Proc. IEEE Workshop on Multimedia Software Development (MMSD'96)*, Berlin, Germany, 25 March 1996.

[BBH97] Bhavesh Bhatt, David Birks & David Hermreck. "Digital Television: Making it Work." *IEEE Spectrum* 34(10), pp 19–28, October 1997.

[BeRi95] Luc Bellissard & Michel Riveill. "Olan: A Language and Runtime Support for Distributed Application Configuration." *Journées du GDR du Programmation*, Grenoble, France, November 1995.

[BlSt98] Gordon Blair & Jean-Bernard Stefani. *Open Distributed Processing and Multimedia*. Addison-Wesley, Harrow, England, 1998.

[FoSl96] Halldor Fosså & Morris Sloman. "Implementing Interactive Configuration Management for Distributed Systems." *Proc. 4th International Conference on Configurable Distributed Systems (CDS'96),* Annapolis, Maryland, USA, pp 44–51, May 1996.

[Gib95] Simon J. Gibbs & Dionysios C. Tsichritzis. *Multimedia Programming: Objects, Frameworks and Environments.* Addison-Wesley, Wokingham, England, 1995.

[GoCr96] Kaveh Moazami Goudarzi & Jeff Kramer. "Maintaining Node Consistency in the Face of Dynamic Change." *Proc. 4th International Conference on Configurable Distributed Systems (CDS'96)*, Annapolis, Maryland, USA, pp 62–69, May 1996.

[HaRe83] T. Härder & A. Reuter. "Principles of Transaction-Oriented Database Recovery." *ACM Computing Surveys* 15(4), 1983.

[Hel94] Tobias Helbig. "Development and Control of Distributed Multimedia Applications." *Proc. 4th Open Workshop on High-Speed Networks*, Brest, France, 7–9 September 1994.

[KaPu91] Gail E. Kaiser & Calton Pu. "Dynamic Restructuring of Transactions." In Ahmed K. Elmagarmid (ed.), *Database Transaction Models for Advanced Applications,* Chapter 8. Morgan Kaufmann, 1991.

[Kin93] Tim Kindberg. "Reconfiguring Client-Server Systems." *Proc. International Workshop on Configurable Distributed Systems* (IWCDS'94), 1994.

[KrMa90] Jeff Kramer & Jeff Magee. "The Evolving Philosophers Problem: Dynamic Change Management." *IEEE Transactions on Software Engineering* 16(11), pp 1293–1306, November 1990.

[Mit98] Scott Mitchell. "Multimedia Transactions in the DJINN Framework." Technical Report, Queen Mary & Westfield College, Department of Computer Science (in progress).

[MNCK97] Scott Mitchell, Hani Naguib, George Coulouris & Tim Kindberg. "A Framework for Configurable Distributed Multimedia Applications." *3rd Cabernet Plenary Workshop*, Rennes, France, 16–18 April 1997.

[SKB88] Janos Sztipanovits, Gabor Karsai & Ted Bapty. "Self-Adaptive Software for Signal Processing: Evolving Systems in Changing Environments Without Growing Pains." *Communications of the ACM* 41(5), pp 66–73, May 1998.

[WGH94] Stuart Wray, Tim Glauert & Andy Hopper. "The Medusa Applications Environment." Technical Report 94.3, Ollivetti Research Limited, Cambridge, England, 1994.