Dragonfly: Linking Conceptual and Implementation Architectures of Multiuser Interactive Systems

Gary E. Anderson **T.C. Nicholas Graham**
Department of Computing and Information Science
Queen's University
Kingston, Ontario, Canada
+1 613 533 6526
look2infinity@geocities.com, graham@cs.queensu.ca

Timothy N. Wright
Department of Computer Science
University of Canterbury
Private Bag 4800
Christchurch, New Zealand
tnw13@cosc.canterbury.ac.nz

ABSTRACT
Software architecture styles for developing multiuser applications are usually defined at a *conceptual* level, abstracting such low-level issues of distributed implementation as code replication, caching strategies and concurrency control policies. Ultimately, such conceptual architectures must be cast into code. The iterative design inherent in interactive systems implies that significant evolution will take place at the conceptual level. Equally, however, evolution occurs at the implementation level in order to tune performance. This paper introduces Dragonfly, a software architecture style that maintains a tight, bidirectional link between conceptual and implementation software architectures, allowing evolution to be performed at either level. Dragonfly has been implemented in the Java-based TeleComputing Developer (TCD) toolkit.

Keywords
Software architecture, user interface development toolkits, groupware

1 INTRODUCTION

The spread of the Internet has led to the emergence of computing applications that support communication and collaboration among people, rather than purely computational tasks. This class of application, often called *groupware* [12], imposes challenging requirements on the software development process. As it is difficult to predict usability problems with groupware systems, their design requires iterations of implementation and testing with users. Resultingly, a system's design may evolve considerably as it is being developed. However, groupware applications are also distributed systems, with exigent performance requirements. Low-level implementation issues such as network communication, caching, replication and concurrency control increase the difficulty of modifying systems once they have been implemented. Therefore, the iterative requirement required to create usable multiuser applications is hard to achieve in practice.

Several architecture styles have been proposed to address the difficulties of iterative refinement of distributed groupware systems. These include PAC* [3], C2 [23], ALV [16], and our own Clock architecture style [13]. These share the property of being *conceptual* styles [20], allowing designers to express a program's high-level structure while abstracting details of its implementation as a distributed system. Conceptual architectures are intended to capture the developer's abstract view of the system, where components correspond directly to features of the application's user interface or functional core. Conceptual architectures can be methodically derived from the user interface design or from task models [19], thus representing an intermediate step between user interface design and system implementation. Additionally, conceptual architectures permit early evaluation of the impact of user interface design decisions on the implementation. For example, Dewan [6] has applied the Software Architecture Analysis Method (SAAM) [2] to show how architectures can expose implementation trade-offs in groupware applications.

Design processes for multiuser interactive systems promote the design of the conceptual architecture following the design of the user interface [5]. Conceptual architectures must then be mapped to concrete *implementation* architectures, where issues of distribution, networking protocols, caching, replication and consistency control are addressed. A wide space of implementation architectures (as characterized by Dewan [6] and Phillips [20]) is available for any given conceptual architecture.

Evolution in a groupware system may impact either the conceptual or the implementation level. Modifications to the system's functionality resulting from usability problems are best addressed in the conceptual architecture, while modifications resulting from performance requirements impact the implementation architecture. Once the application has been mapped to an implementation architecture, however, the link to the conceptual

architecture may be lost. To address this problem, we present *Dragonfly*, a novel software architecture style for synchronous groupware systems. Dragonfly subsumes both conceptual and implementation architectures, allowing developers to smoothly move between levels, as appropriate to the development or evolution task. Dragonfly is based on an orthogonal decomposition of application functionality and distribution issues, and permits a wide range of implementation architectures.

Dragonfly differs from the three basic approaches currently used to map conceptual architectures for groupware to implementation architectures. These are:

- *Principled mapping:* Implementation architectures can be derived from conceptual architectures through the methodical application of mapping guidelines. For example, Duval and Nigay demonstrate a method for mapping PAC-Amodeus architectures to Java implementations [9].
- *Tool mapping:* A second approach is to build the mapping directly into a tool. Tools such as Suite [7], RendezVous [16], Groupkit [21] and Clock [13] permit developers to work with conceptual architectures based on the Model-View-Controller style [17], while the tool automatically creates a distributed implementation.
- *Open implementation:* A variant combining the flexibility of mapping by hand and the convenience of a toolkit is the open implementation approach, where the mapping provided by the toolkit can be customized either through high-level parameters [7, 24], through meta-languages controlling the operation of the toolkit's runtime system [8, 18, 22] or through opportunities for inserting code directly into the toolkit [10].

Each of these approaches has disadvantages, either in traceability, flexibility or ease of use. Hand mapping leaves no explicit link between conceptual and implementation architectures, making future evolution at the conceptual level difficult. Tools maintain the link, but typically provide a restricted choice of implementation architectures. Open implementation tools provide greater flexibility, but require developers to understand the detailed internal operation of the development tool.

Dragonfly differs from these approaches by establishing a one-to-one mapping between components of the conceptual and implementation architectures. This mapping allows the conceptual architecture to be easily retrieved from the implemention architecture, allowing future evolution to proceed at either level. Dragonfly provides an orthogonal treatment of the implementation of application functionality, code and data distribution, replication, caching, and concurrency control. This orthogonality permits a wide range of distributed implementations from a single conceptual architecture. Dragonfly has been implemented in the Java-based TeleComputing Developer toolkit [1].

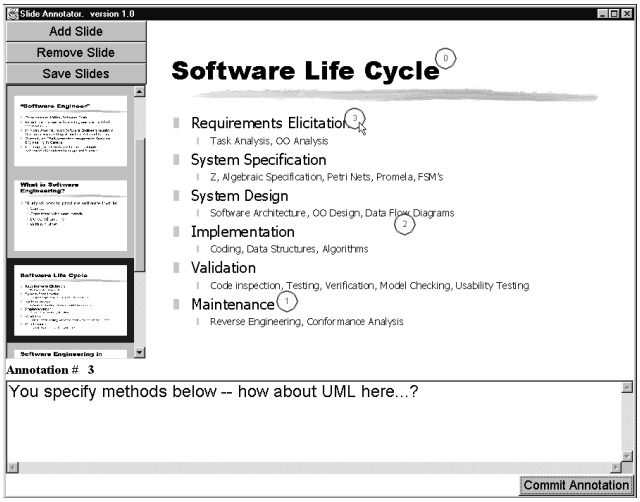

Figure 1: Slide Annotator.

The paper is organized as follows. Section 2 discusses the roles of conceptual and implementation architectures in the development of groupware, and motivates the need for a tight coupling between them. Section 3 introduces the Dragonfly architecture, and explains how it maintains this tight coupling while presenting a high degree of flexibility in the implementation architecture. Section 4 discusses our implementation of Dragonfly in the TeleComputing Developer toolkit, followed by a discussions of the advantages and disadvatages of this approach in section 5.

2 CONCEPTUAL AND IMPLEMENTATION ARCHITECTURES

In order to motivate the difference between conceptual and implementation architectures, we introduce an example, a multiuser slide annotation program (see figure 1).[1] While simple, this application serves to illustrate the relation between conceptual and implementation architecture, and the need to evolve at both levels.

The slide annotator permits a small group of people to view and discuss a set of presentation slides, and annotate the slides for future reference. The set of slides appears in thumbnail form to the left of the currently viewed slide. The current annotation appears below the slide.

To add an annotation, a user clicks on the slide surface, creating a circle indicating the position and number of the annotation. The text for the annotation can be

[1]The user interface of this slide annotator was based on a similar system presented by Li and Muntz [18].

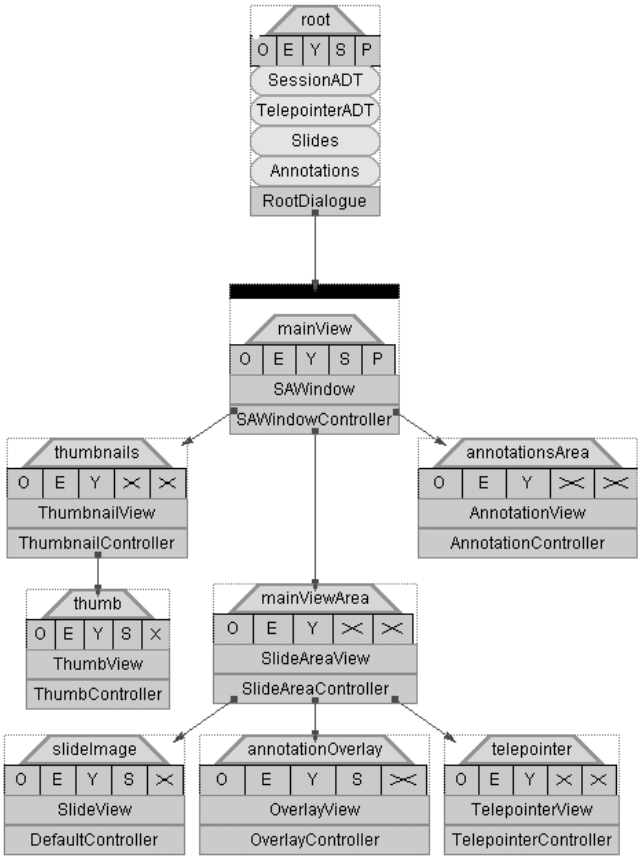

Figure 2: Conceptual architecture of slide annotator from figure 1, expressed using the TCD Toolkit.

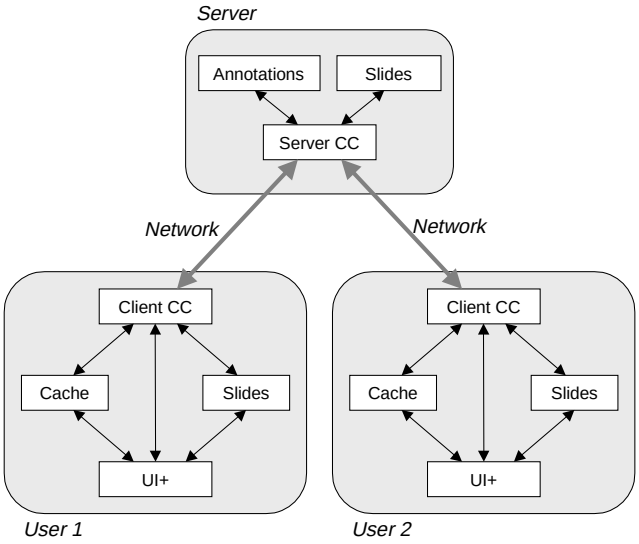

Figure 3: Client/server implementation architecture.

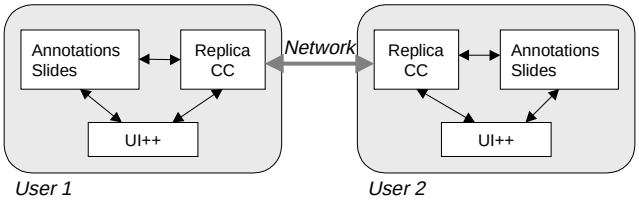

Figure 4: Replicated implementation architecture.

filled in using the text box below the slide. Clicking on an annotation circle displays the annotation for that circle, allowing it to be viewed or edited.

All users' views of the slide annotator are synchronized. That is, if one user clicks a thumbnail to change which slide is being viewed, all users' views will change. If one user adds a new annotation, all users see the annotation in real time.

Conceptual Architecture

Figure 2 shows the conceptual architecture of the slide annotator, expressed in the Clock architecture style [13] using the TeleComputing Developer toolkit. The Clock architecture style is similar to PAC [3], ALV [16], and C2 [23] in that it encourages developers to structure their applications hierarchically, based on the compositional structure of the user interface. (The Clock architecture style can be seen as a layered extension of the Model-View-Controller architecture [17].) Here, the main view (the *mainView* component) is composed of the set of *thumbNails*, the main slide view (*mainViewArea*) and the current annotation (*annotationsArea*). One instance of this *mainView* is created for each user.

Shared data is represented in the *root* component. For example, the *SessionADT* contains information about people currently in a slide annotation session. The *Slides* ADT contains the information about the slides themselves (actually, the location where the slides are stored as GIF images). The *Annotations* ADT contains the locations and text of the annotations for each slide.

The complete conceptual architecture shows the application is decomposed into components, each responsible for maintaining data shared between users or for maintaining some part of the screen. The architecture specifies how these components communicate, and provides temporal guarantees ensuring consistency control over shared data [13].

Implementation Architectures

Conceptual architectures do not describe how a system is actually implemented as a distributed system. Before implementation, the developer must decide how the code and data are to be distributed over the users' machines, what protocols are to be used for these machines to communicate, what concurrency control algorithms

are to be used, and what data is to be replicated or cached.

A wide space of possible implementation architectures exist for a given conceptual architecture [6, 20]. We give two examples of this space here, and in section 3 discuss the actual space of implementation architectures addressed by Dragonfly.

Figure 3 shows a client/server implementation of the conceptual architecture. Here, the shared data (the slides and annotations) are represented on a server. The user interfaces (*UI+*) for each session participant are represented on their own machines. Concurrency control components are added to the server and client to avoid race conditions. For efficiency, the *Slides* ADT is replicated to each client, while the *Annotations* are represented centrally, but cached on the client machines.

Figure 4 shows an alternative implementation, where both the annotations and slides are replicated, and where a concurrency control component (*Replica CC*) is responsible for multicasting updates, and detecting and undoing conflicting actions.

The user interface components in these examples (*UI+* and *UI++*) consist of the *mainView* tree from figure 2, modified for the particular implementation. For example, in figure 3, the *UI+* component must synchronize with the client concurrency control before, for example, updating its display. On the other hand, in figure 4, the *UI++* component must be constructed to permit some form of undo or operation transform [11] facility on shared data in case of conflicting operations.

Both of these architectures have advantages over the other. For example, as described by Dewan [6], the client/server variant makes it easier for late-comers to enter a session, while the fully replicated version may have better performance in very poor latency environments.

These examples illustrate the benefit of separating the conceptual architecture (describing the implemenation of an application's functionality) from the implementation architecture (describing distributed systems issues). This separation allows developers to concentrate on a clear functional decomposition of a system separately from complex low-level issues such as caching and concurrency control strategies.

Evolution

Applications may evolve both at the level of their functional design (as represented by their conceptual architecture) and their distributed implementation. For example, if we wish to add a feature to the slide annotator allowing free-hand drawing of annotations on the slide, we would like to return to the conceptual architecture to make the change. However, if we determine that application performance is poor, we may wish to modify the distributed implementation architecture.

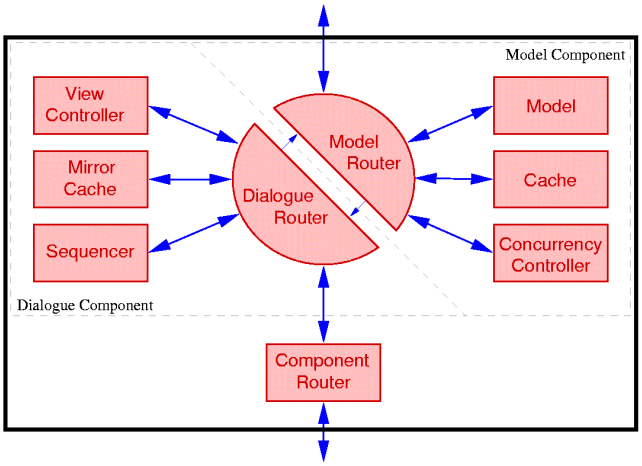

Figure 5: The Dragonfly component and its implementation facets.

Evolution at each level cannot, however, be performed in isolation. For example, changing the conceptual architecture may impact the implementation architecture by imposing new performance requirements. Similarly, improving the performance of the implementation architecture might make it possible to implement application features that were not possible earlier.

As outlined in the introduction, current approaches do not provide good support for this tightly-coupled evolution at the conceptual and implementation level. If the implementation architecture has been created by hand from the conceptual one, the conceptual architecture may be for all practical purposes lost, requiring all future evolution to take place at the implementation level. Significant evolution at the implementation level may be very hard – for example, changing from client/server to full replication may require significant reprogramming. If a tool is used to hide the implementation architecture, evolution at the conceptual level is simpler, but evolution at the implementation level may be either restricted or impossible.

In the next section, we present *Dragonfly*, an architecture style designed to maintain a tight link between conceptual and implementation architectures for distributed multiuser applications.

3 DRAGONFLY

Dragonfly is an architecture style for developing distributed multiuser applications. Dragonfly is designed to support evolution at the conceptual and the implementation level, by allowing developers to move smoothly between the levels. In order to achieve these

goals, Dragonfly provides a bidirectional link between the conceptual architecture and its distributed implementation, and simplifies the modification of the implementation architecture. These properties are obtained by following the principles of:

Orthogonality: The Dragonfly component treats issues of distributed implementation orthogonally from issues of the implementation of application's functionality. In particular, replication, caching, concurrency control, application model, view and controller are all implemented as separate *facets* of the Dragonfly component.

Traceability: Dragonfly components map one-to-one with components in the conceptual architecture. This allows the conceptual architecture to be retrieved from the implementation architecture by projecting those parts that implement the application functionality.

Replaceability: Mechanisms for distributed implementation can be plug-replaced.

Flexibility: Dragonfly permits a wide range of distributed implementations.

The following sections shows how these properties are implemented in Dragonfly, and how they contribute to our goals of permitting smooth, bidirectional transition between conceptual and implementation architecture. Section 4 then shows how Dragonfly is realized as part of the TeleComputing Developer groupware development toolkit. Section 5 describes the lessons learned from our implementation of Dragonfly, and discusses the benefits and limitations of this approach.

Orthogonality

The primary property supporting Dragonfly's tight linkage of conceptual and implementation architectures is the orthogonal treatment of implementation issues. As shown in figure 5, the Dragonfly component provides separate *facets* responsible for implementing the application's functionality, as well as concurrency control, data replication and caching. Additionally, components can be located together on one machine, or distributed across a network. As we shall see in the next sub-section, this orthogonality allows developers to smoothly move between the conceptual and implementation architectures, working at whichever level is appropriate to the task.

Additionally, the orthogonality property allows developers to easily change one implementation policy without affecting others. For example, the concurrency control policy for part of the architecture can be changed without necessarily requiring a change in caching policies. Under "Replaceability" below, we describe how this property is obtained in Dragonfly.

In addition to the six implementation facets described below, Dragonfly uses three *router* facets to route messages between dialogue facets, model facets and other components. Similarly to C2 [23], Dragonfly components have a notion of a *top*, which can be used to obtain model data from higher in the architecture, and a *bottom*, which can be used to obtain subviews created by components lower in the architecture.

The dialogue part of the Dragonfly component consists of three facets:

View/Controller: The view/controller implements interaction with the user: a *view* is responsible for maintaining the display, while a *controller* is responsible for processing user inputs. These may be separate objects, or may be implemented together. The view/controller implements interaction functionality determined in the conceptual architecture.

Sequencer: The sequencer is responsible for dialogue-side concurrency control. That is, the sequencer ensures that race conditions do not occur while the view is being recomputed, or while user inputs are being processed.

Mirror Cache: The mirror cache optionally keeps track of requests made to this component. This functionality is useful in implementing server side caching schemes.

The model is responsible for maintaining the data on which the user interface is based. In the slide annotator, the names of the slides and the contents of the annotations are examples of model data.

Model: The model facet itself contains a set of ADT's from which the model is constructed. These ADT's may be replicas of ADT's represented in other Dragonfly components.

Concurrency Controller: The concurrency controller is responsible for sequencing requests and updates to the model so that race conditions do not occur.

Cache: The cache facet optionally records requests originating from this component, and the responses returned by other components. The cache is the dual of the mirror cache facet described above: the mirror cache records requests *made to* this component, while the cache records requests *originating from* this component.

Traceability

Dragonfly's facet decomposition permits a one-to-one mapping between components of the conceptual and implementation architectures, where the implementation architecture is realized by appropriate facet implementation. This one-to-one mapping is the key to allowing easy transition between conceptual and implementation architectures.

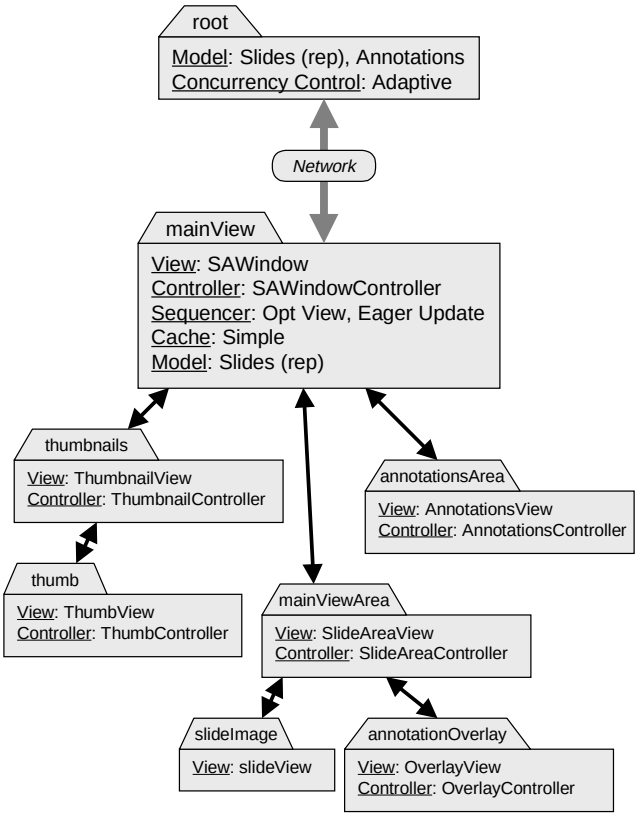

Figure 6: The client/server architecture of figure 3 rendered as a Dragonfly architecture. Facets with null value are ommited.

Figure 6 sketches how the conceptual architecture of figure 3 is implemented in Dragonfly. Here, the root component is represented on a server machine, while each instance of the user interface is connected to this server via a network. The model part of this component contains the *Slides* and *Annotations* ADT's, where *Slides* is marked as being replicated. A concurrency control facet is specified to avoid race conditions in the actions of the different clients. The server has no caches, view or controller, so these facets are filled in with place-holder (or *null*) facets.

The *mainView* component contains a view (*SAWindow*) and controller (*SAWindowController*), as well as a sequencer that provides optimistic update and view computation. A simple cache is provided to cache the annotations from the server. The model contains a replica of the *Slides* ADT.

The remaining user interface components contain the appropriate views and controllers.

When compared to figure 2, we see that there is indeed a one-to-one correspondence between the components in

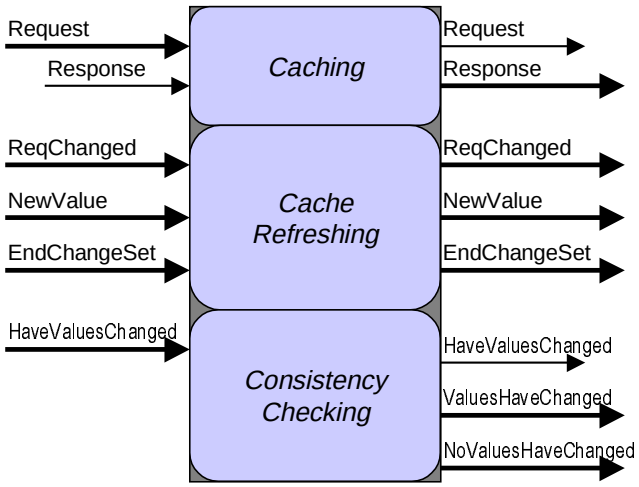

Figure 7: The Dragonfly cache facet.

the Dragonfly architecture and those of the conceptual architecture. This correspondence is the basis of our ability to move between conceptual and implementation levels. The implementation architecture maintains the structure of the conceptual architecture: simply by hiding all facets other than the model, view and controller, the conceptual architecture can be retrieved from the implementation architecture.

This correspondence supports the restructuring of the implementation architecture while maintaining the structure of the conceptual architecture. For example, to move towards the implementation architecture of figure 4, we might modify the model facet of the root component to replicate the *Annotations* ADT, and remove the cache in the mainView.

The key to being able to easily map conceptual architectures to distributed implementations by simple changing the contents of Dragonfly component facets is the design of the facets themselves. Facets must be *plug-replaceable*, so that any facet implementation can be interchanged with any other valid implementation of the same facet type. Replaced facets may not affect the correct operation of other facets, located in the same or other components. Furthermore, the definition of facet interfaces must be sufficiently *flexible* that a wide range of implementations can be encoded by the Dragonfly architecture.

Replaceability

Dragonfly facets must be coded to conform to the Dragonfly protocol [1], a messaging protocol allowing inter-facet and inter-component communication. Facets must implement a specified message interface, subject to a required minimal functionality that guarantees the correct operation of all the facets together.

Resultingly, a facet can be plug-replaced by any other facet without compromising the correctness of other facets in the architecture. This allows, for example, the application's concurrency control strategy to be modified independently of its caching strategy.

Developers using the Dragonfly architecture do not normally need to know of the existence of the Dragonfly protocol. As shown in section 4, developers can create implementation architectures by assembling pre-defined facets that implement a wide range of distribution policies.

The full details of the Dragonfly protocol and its associated messages, facet interfaces and facet required minimal functionality are documented elsewhere [1]. Here, we use the example of the cache facet to give a flavour of the architecture.

Figure 7 shows the interface of the cache facet. In this figure, arrows on the left show the messages that the component must handle, and entries on the right show the messages the component may send. The most important messages are highlighted with bold arrows. This facet has three core functions: to cache the values of request messages, to refresh these values when they become stale, and to permit consistency checking.

Caching: The main purpose of a cache is to take *Request* messages, and respond to these requests with *Response* messages. A request is similar to a method call: it has a name and parameters, and returns some result. If the cache cannot respond to the request itself, it must reissue the request, and pass on the response when it arrives.

Cache Refreshing: When data on which the cache depends changes, the cache may receive two kinds of messages. The *ReqChanged* message indicates that the value of some request has changed; i.e., that any cache entries involving this request are now stale. The *NewValue* message indicates that the value of some *specific* request has changed to a given new value. The cache may deal with these messages locally, or forward them.

Consistency Checking: Some concurrency control algorithms rely on being able to determine whether data used in computing a transaction has since become out of date. The *HaveValuesChanged* message carries a set of request/response pairs, and requires a message in response (*ValuesHaveChanged* or *NoValuesHaveChanged*).

Any cache implementing this interface can be plugged into this facet. Even though the cache implicitly depends on changes made in model facets, the cache does not need to be aware of how these changes occur, and therefore can be orthogonally combined with any implementation of other facets.

Based on this interface, we have implemented three caches: a *null cache* that performs no caching at all, a *simple cache* that caches the values of all request messages flowing through it, and a *prefetch cache* [14] that actively updates stale cache entries before they are required.

Each Dragonfly facet is described through an interface, which is obliged to provide a minimal functionality in order to guarantee correct interoperation with other facets [1].

Flexibility

In addition to being plug-replaceable and orthogonally combinable with other facets, it is important that facet interfaces be sufficiently flexible to allow a wide range of implementations.

As motivated by Dewan's reference architecture for groupware [6] and our own work on flexible implementation architectures [14, 24], we designed Dragonfly to be flexible in three dimensions:

Concurrency control: A full range of *pessimistic* and *optimistic* strategies [15] should be supported.

Replication: It should be possible on a per-ADT basis to choose between centralized and replicated implementation of shared data.

Caching: A full range of caching strategies, located at the client and server, either active or passive [14], should be possible.

Additional dimensions for future consideration include security and fault tolerance.

We have already seen how the definition of the cache facet permitted three implementations with different properties. As described in section 4, the TeleComputing Developer implementation of Dragonfly supports a large subset of the space. Section 4 describes the facet implementations we have created in Dragonfly.

4 THE TELECOMPUTING DEVELOPER

In order to demonstrate the practicality of the Dragonfly architecture, we have constructed the *TeleComputing Developer* (TCD), a Java-based toolkit for groupware development. TCD provides an implementation of the Dragonfly component, and a set of implementations of the Dragonfly facet supporting a wide range of distributed implementations. TCD implements the full Dragonfly protocol. TCD provides a high-level interface to the Dragonfly architecture, hiding from the developer the details of facet implementation and the Dragonfly protocol itself.

The Dragonfly component is presented to developers as a visual bean. This allows developers to create and edit the conceptual architecture of their system by direct manipulation, using a standard interface builder. Fig-

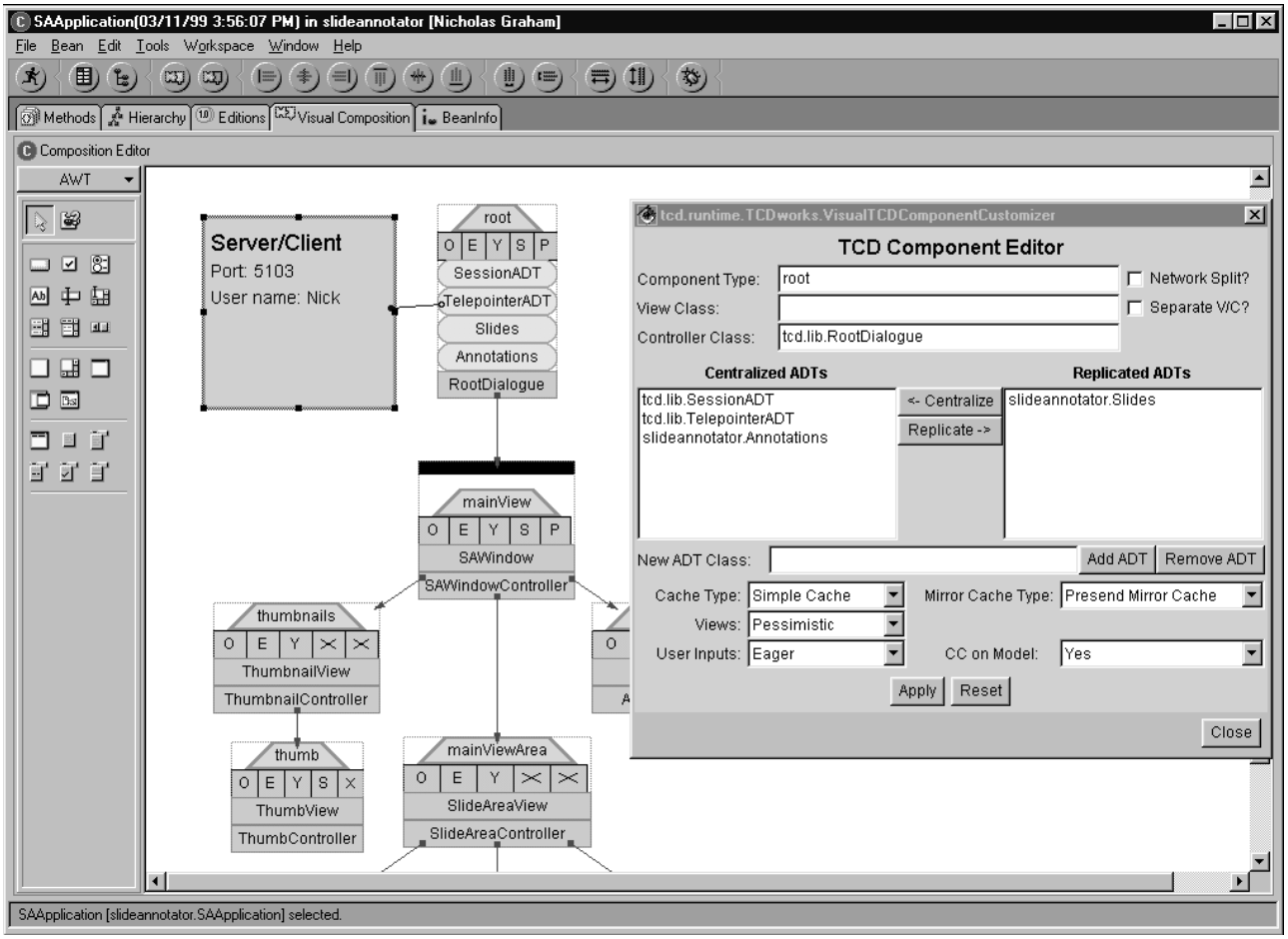

Figure 8: Component customization in the TeleComputing Developer.

ure 8 shows the use of IBM's VisualAge for Java [4] to construct the slide annotator architecture from figure 2.

The conceptual architecture is mapped to the implementation architecture by manipulating properties in a component editor (also shown in figure 8.) For each component, the developer can select from a list of predefined facet implementations, can add and replicate ADT's, and can specify network splits. Therefore, the developer does not need to be aware of the details of the Dragonfly protocol or of the structure of the Dragonfly component. Advanced developers may of course choose to create their own facet implementations for special purposes.

TCD shows that Dragonfly achieves the objective of permitting smooth transition between conceptual and implementation architecture. The conceptual architecture is always visible. To make changes at the implementation level, the developer simply uses the component editor to plug-replace facets in one or more components.

In order to demonstrate the flexibility and pluggability of Dragonfly facets, TCD provides a wide range of facet implementations. These are summarized in figure 9. From this list, it can be seen that a wide range of implementation architectures can be derived from a single conceptual architecture.

The facet implementations of figure 9 cover a large subset of the space of distributed implementations identified in section 3. In particular, the currently implemented facets support:

Concurrency control: Both optimistic and pessimistic concurrency control are supported. However, optimistic schemes are based on rollbacks. The protocol would have to be extended to support the operation-transform style [11] of optimistic concurrency control.

Replication: ADT's may be replicated on a per-ADT basis. However, inter-replica communication is always performed through a single machine, rather

Facet	Implementation	Description
View/Controller	ViewController	Standard implementation allows developer to specify view and controller object, either of which may be null.
Mirror Cache	Null	No mirror cache.
	Presend	Mirror cache records requests made from lower components, actively sends *NewValue* messages to inform clients of changes.
Sequencer	Null	No sequencing – may result in concurrency errors.
	Pessimistic	Sequencer obtains locks before processing inputs or computing view.
	Optimistic	Sequencer optimistically processes inputs or computes views.
Model	Model	Standard implementation allows developer to specify ADT's in model, and to optionally tag them as replicated.
Cache	Null	Performs no caching.
	Simple	Caches requests and response values. Refreshes stale entries when they are requested again.
	Prefetch	Caches requests and response values, and refreshes stale entries whether they are requested or not.
Concurrency Controller	Null	Provides no concurrency control – may result in concurrency errors.
	Adaptive	Performs either pessimistic or optimistic concurrency control, automatically adapting to actions of sequencer facet.

Figure 9: Facet implementations provided in TCD.

than through direct multicasting.[2]

Caching: The full range of client and server caches, both active and passive, has been implemented.

Therefore, Dragonfly has proved sufficiently flexible to implement a large subset of the space of implementation architectures. We anticipate that only minor modifications to the Dragonfly protocol will be required to support the parts of the implementation space that are not yet covered.

5 ANALYSIS

Our experience with the TCD toolkit indicates that Dragonfly meets its goal of supporting smooth, bidirectional transition between conceptual and implementation architectures, while being sufficiently flexible to allow a wide range of distributed implementations. We now consider the advantages and disadvantages of this approach for software architecture design.

The Costs of Flexibility

We have built considerable flexibility into Dragonfly's components, in order to permit the wide range of facet implementations seen in figure 9, and others that we may not have anticipated. There is a fine balance in choosing the correct degree of flexibility. Too little flexibility implies that some facet implementations may not be possible. Too much flexibility implies that the facets themselves may become too hard to implement. Furthermore, flexibility in one facet may impact the minimal required functionality in other facets, making all facets harder to implement.

[2]The TCD implementation of ADT replication is incomplete. Currently, the appropriate replication messages are sent, but ignored.

Orthogonality

The goal of orthogonality in the architecture requires that any combination of facets work together. Some facet combinations, however, while semantically correct give poor performance. For example, a presend mirror cache provides optimizations to refresh client side caches quickly. A presend mirror cache simply brings overhead to the system if no client caches are in place (and should be replaced by a null mirror cache.) It would be interesting to investigate higher level mechanisms for tuning the distribution architecture that aid the developer in choosing sensible facet combinations.

Performance

Dragonfly's flexibility requires the overhead of sending messages between facets that are sometimes not required for the particular facet implementations being used. Based on our earlier work showing that significant performance gains are possible by tuning the implementation architecture of groupware applications [24], we anticipate that these overheads will prove small compared to the benefits of tuning. However, future work in tuning Dragonfly and performing performance evaluation will be required to validate this hypothesis.

6 CONCLUSIONS

This paper has presented Dragonfly, a software architecture supporting the development of synchronous groupware applications. The paper has shown how Dragonfly addresses the problem of permitting evolution of groupware applications to occur at both the levels of the conceptual architecture and the distributed implementation architecture. This is achieved by the orthogonal treatment of distribution issues from the functionality of the application itself. We have shown that this orthogonality can be achieved by splitting the Dragonfly component into a set of facets, each responsible for one aspect of the distributed implementation architecture. We have argued that these facets need to be designed to be orthogonal, traceable, replaceable and flexible.

ACKNOWLEDGEMENTS

This work was performed at the Software Technology Laboratory at Queen's University, under the support of the National Science and Engineering Research Council (NSERC). We gratefully acknowledge many helpful discussions with Greg Phillips and the members of IFIP WG 2.7 on User Interface Engineering. Thanks are due to Laurence Nigay for critical reading of a draft of this paper.

REFERENCES

[1] G.E. Anderson and T.N. Wright. TeleComputing Developer implementation design. http://stl.cs.queensu.ca/~tcd/Design.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[3] G. Calvary, J. Coutaz, and L. Nigay. From single-user architectural design to PAC*: A generic software architecture model for CSCW. In *Proc. CHI '97*, pages 242–249. ACM Press, 1997.

[4] IBM Corporation. VisualAge for Java. http://www.software.ibm.com/ad/vajava.

[5] J. Coutaz. PAC-ing the architecture of your user interface. In *Proc. DSV-IS '97*, pages 15–32. Springer Verlag, 1997.

[6] P. Dewan. Architectures for collaborative applications. In M. Beaudouin-Lafon, editor, *Computer Supported Co-operative Work*. John Wiley & Sons Ltd., January 1999.

[7] P. Dewan and R. Choudhary. Coupling the user interfaces of a multiuser program. *ACM TOCHI*, 2(1):1–39, March 1995.

[8] P. Dourish. Using metalevel techniques in a flexible toolkit for CSCW applications. *ACM TOCHI*, 5(2):109–155, 1998.

[9] T. Duval and L. Nigay. Implémentation d'une application de simulation selon le modèle PAC-Amodeus. In *Proc. IHM '99*, 1999.

[10] W.K. Edwards, E.D. Mynatt, K. Petersen, M.J. Spreitzer, D.B. Terry, and M.M. Theimer. Designing and implementing asynchronous collaborative applications with Bayou. In *Proc. ACM UIST '97*, pages 119–128. ACM Press, 1997.

[11] C.A. Ellis and S.J. Gibbs. Concurrency control in groupware systems. In *Proc. SIGMOD '89*, pages 399–407. ACM Press, 1989.

[12] C.A. Ellis, S.J. Gibbs, and G.L. Rein. Groupware: Some issues and experiences. *CACM*, 34(1):38–58, January 1991.

[13] T.C.N. Graham and T. Urnes. Integrating Support for Temporal Media into an Architecture for Graphical User Interfaces. In *Proc. ICSE 19*, pages 172–182. ACM Press, 1997.

[14] T.C.N. Graham, T. Urnes, and R. Nejabi. Efficient distributed implementation of semi-replicated synchronous groupware. In *Proc. ACM UIST '96*, pages 1–10. ACM Press, 1996.

[15] S. Greenberg and D. Marwood. Real time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proc. ACM CSCW '94*, pages 207–217. ACM Press, 1994.

[16] R.D. Hill, T. Brinck, S.L. Rohall, J.F. Patterson, and W. Wilner. The *Rendezvous* language and architecture for constructing multi-user applications. *ACM TOCHI*, 1(2):81–125, June 1994.

[17] G.E. Krasner and S.T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *JOOP*, 1(3):26–49, August/September 1988.

[18] D. Li and R. Muntz. COCA: Collaborative objects coordination architecture. In *Proc. ACM CSCW '98*, pages 179–188, 1998.

[19] F. Paterno, C. Mancini, and S. Meniconi. Engineering task models. In *Proc. IEEE Conference on Engineering Complex Systems*, pages 69–76. IEEE Press, 1997.

[20] W.G. Phillips. Architectures for synchronous groupware. Technical Report 1999-425, Department of Computing and Information Science, Queen's University, May 1999.

[21] M. Roseman and S. Greenberg. Building real time groupware with GroupKit, a groupware toolkit. *ACM TOCHI*, 3(1):66–106, March 1996.

[22] F. Tarpin-Bernard, B. David, and P. Primet. Frameworks and patterns for synchronous groupware: AMF-C approach. In *Proc. EHCI '98*, pages 225–242, September 1998.

[23] R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead Jr., J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Trans. SW Eng.*, 22(6):390–406, June 1996.

[24] T. Urnes and T.C.N. Graham. Flexibly mapping synchronous groupware architectures to distributed implementations. In *Proc. DSVIS'99*, pages 133–148, 1999.