Workspaces: A Multi-level Architectural Style for Synchronous Groupware

W. Greg Phillips[1] and T.C. Nicholas Graham[2]

[1] Royal Military College of Canada, Kingston, Ontario, Canada K7K 7B4
Department of Electrical and Computer Engineering
greg.phillips@rmc.ca
[2] Queen's University, Kingston, Ontario, Canada K7L 3N6
School of Computing
graham@cs.queensu.ca

Abstract. We present a new architectural style for synchronous groupware that eases the transition from scenario based modeling to component design, and from component design to distributed implementation. The style allows developers to work at a distribution-independent conceptual level and provides for automatic or semi-automatic refinement of conceptual designs into appropriate distributed implementations at run-time. Both the conceptual and implementation levels of the system can be evolved dynamically at run-time in response to user needs and changes in the distributed system environment. System evolution at both levels is specified via an evolution calculus.

1 Introduction

The enhancement of interpersonal communication and collaboration was one of the goals driving development of the network that eventually became the Internet [9]. J.C.R. Licklider's vision for a large-scale network included specialized software supporting both asynchronous and synchronous ("near real time") collaboration — software that we now call *groupware* [6]. There are many examples of successful asynchronous collaborative tools, ranging from email to weblogs to distributed source code management systems. However it is only recently, with the increasing availability of relatively high bandwidth, low latency, always-on network connections, that we have begun to achieve some of the tantalizing potential of synchronous groupware.

For synchronous groupware systems (hereafter simply "groupware") to be usable and effective, the development of groupware must be informed by the ways in which people actually work and play together. Studies of real-world collaboration confirm what we understand intuitively: that people move fluidly between individual and collaborative activities, that collaborations frequently incorporate a variety of tool sets, and that people are often involved in a mix of several concurrent individual and collaborative activities [4]. By contrast, most current groupware tools embed collaboration within distinct applications. People who wish to collaborate at a

distance must start a groupware application, interact using the application, and then end the collaboration by closing the application [11, 16].

We believe that the clear mismatch between natural collaboration styles and application-centric collaborative systems may explain why there are so few successful synchronous groupware systems, outside of specialized areas such as online gaming and distributed meeting support. Further, the application-centric approach ignores the fact that the users' expectations of the collaborative system, as well as the network infrastructure supporting communication and collaboration, are constantly in flux. Users may initiate, terminate, join and leave collaborative activities at any time, with or without warning. By the same token, network nodes and links will have different and time-varying performance characteristics and may become saturated or fail outright. Since the requirements of the users and the properties of the networks are ever-changing and prone to induce faults, groupware systems' run-time and distribution architectures must be both dynamic and fault tolerant. We are not aware of any existing groupware systems that allow an approximation of a natural collaboration model [11].

In our view an appropriate architectural approach would invert application-centricity and embed groupware tools within the context of users and their collaborations, just as in the physical world. Shifts between individual and collaborative work would not require users to change either user interfaces or work styles, except where dictated by the nature of the collaboration itself. The architecture would support the development of systems in which users create and dissolve collaborations fluidly, using whatever tools are most appropriate to a given situation. Scenario-based modeling of user requirements would lead naturally to component-level design, and component-level design would not unnecessarily constrain distributed implementation. Finally, changes in the network infrastructure supporting distributed collaborations would be handled gracefully, whether the changes were intentional or accidental, qualitative or catastrophic.

The balance of this paper presents the Workspace Model: a user-centred architectural style for groupware that supports achievement of the goals described above. In section 2 we provide a high-level overview of our approach. This is followed in section 3 by the description of a realistic groupware usage scenario, which is modeled using the *conceptual level* of the Workspace Model. This scenario is used to make concrete and motivate subsequent discussion. Section 4 briefly describes how information at the conceptual level can be used to ease the transition to *component and connector level* design and implementation. In section 5 we extract a portion of the scenario from section 3 and illustrate how the workspace run-time system can automatically and dynamically map the required components and connectors onto distributed system implementations at the Workspace *implementation level.* We have developed two implementations of the Workspace Model and its run-time system, one in Python and one in C++; in section 6 we present the current status of our implementations.

2 Workspaces: A User-centred Approach to Implementation

The Workspace Model has been developed to provide explicit support for four key activities in the design and implementation of groupware systems. These are:

1. scenario-based modeling of intended system use;
2. definition of the system-level properties of components and connectors required to realize the scenario;
3. simple development of the components themselves; and
4. automatic or semi-automatic deployment of connectors and components using a distributed system architecture appropriate to run-time conditions.

While the use of the Workspace Model is intended to be process-neutral, it is most easily understood as a sequence of development phases carried out in the order listed above. This section provides a brief summary of the Workspace approach in terms of these phases. Each phase is elaborated in more detail in the balance of the paper, with a particular focus on the modeling and run-time implementation phases.

Scenario modeling. Collaborative scenarios of interest are modeled using a high-level graphical language called the workspace *conceptual-level notation*, in conjunction with an architectural composition language called the *evolution calculus*. The conceptual notation, which is similar to ASUR [5], depicts run-time snapshots of the scenario at times of interest. Each snapshot corresponds to a scenario state. The evolution calculus provides for precise specification of dynamic behavior during the course of the scenario.

System-level design. Scenario modeling supports the identification of the conceptual-level *components* and *connectors* required to realize the scenario. The topologies arising in scenario diagrams provide clues as to the *ports* that must be supported by the components, the *vocabularies* shared by connectors and ports, and the requirements for shared data to support collaboration.

Component development. Once components and their required ports have been specified they may be implemented in a relatively straightforward fashion — in particular, component implementations may be written without regard to thread-safety. This is possible since the *workspace run-time system* guarantees their protection from concurrent access, even though the workspace model is inherently multi-threaded.[1] Component developers are also freed from responsibility for asynchronous event delivery and the mechanisms used to support shared state, since these are also provided in a flexible fashion by the run-time system. In effect, developers implement components directly at the conceptual level of the Workspace Model, which allows the code to more closely resemble scenario-based design concepts.

While it is also possible to develop specialized connectors within the system, for most purposes we expect that the connector implementations provided by the workspace run-time system will be adequate.

[1] Explicitly multi-threaded components may also be written; however, this is often not necessary.

Run-time implementation. During execution, the workspace run-time system interprets the same evolution calculus operations used for scenario modeling in order to generate and evolve any desired conceptual-level configuration of components and connectors. The calculus allows components to be dynamically created, destroyed, migrated from one workspace to another and attached to and detached from one another. It also allows shareable state to be rendered private, or else shared in a controlled fashion.

As the conceptual-level evolution calculus operations are executed, the run-time system automatically generates a corresponding implementation-level architecture. Implementations combine the scenario-specific components discussed above with a plug-replaceable suite of special purpose support components provided by the run-time system. These components deal with issues such as network communication, concurrency control, replica consistency maintenance, and asynchronous event broadcasting. This approach is similar to the techniques employed in Clock [15] and Dragonfly [2]; however, where Clock and Dragonfly deal only with static architectures, the workspace runtime is fully dynamic.

3 Scenario-based Modeling

One premise underlying the Workspace Model is that we can build better collaborative systems by starting our design from scenario-based models representing the activities the system is to support. In general this presumes an understanding of system context and requirements — which could be intuitive or derived from the application of a complementary technique such as Groupware Task Analysis [17] or ConcurTaskTrees [10]. In this section we first present an overview of our scenario modeling approach, then illustrate it with a relatively simple collaborative scenario.

3.1 Modeling Concepts

Scenario-based modeling is performed using a simple graphical formalism that represents snapshots of the collaboration at moments of interest. These snapshots are called *workspace diagrams* and are expressed using the conceptual-level notation shown in figure 1. The elements of the workspace notation are described later in this paper, as they are introduced.

A workspace diagram represents users, the physical and virtual entities in their environments, and their collaborations with others. The top-level organizational concept in a workspace diagram is the *workspace*, which is a collection of entities used as resources by one or more people in carrying out some task. These entities may be purely physical, like a book or a pen; purely virtual, like a graphical user interface or a database; or they may be *adapters* that translate between the physical and virtual environments, like mice, cameras, displays, and so on. In this regard a workspace diagram is similar to ASUR [5]; however, where ASUR groups the virtual entities into a single “system” component, a workspace diagram decomposes the system into some number of lower-level components and connectors. Workspace diagrams also differ from ASUR in providing a notation for *synchronization*, which is an

implementation-independent representation of state shared between workspaces, and in having explicit notions of evolution over time and of refinement from the conceptual level to the implementation level.

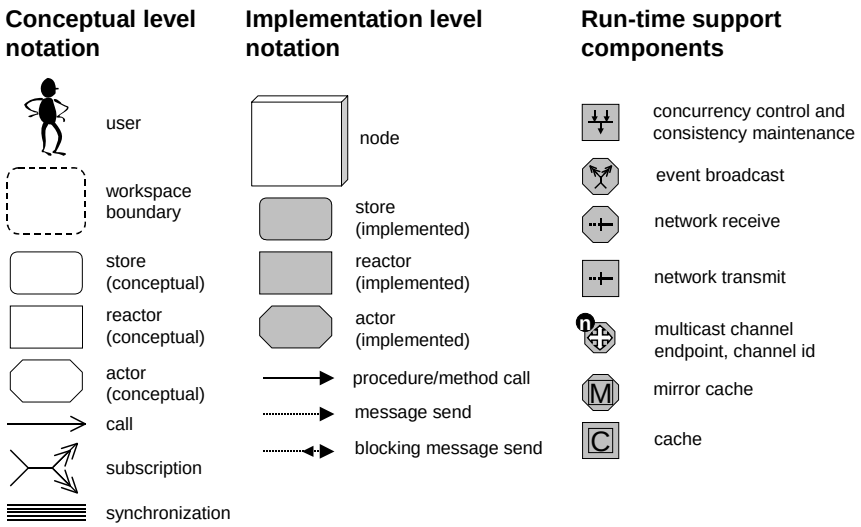

Fig. 1. Workspace notation. White components and open arrowheads are at the conceptual level. Shaded components and filled arrowheads are at the implementation level.

A conceptual model of a scenario consists of a time-sequence of workspace diagrams plus supporting narrative. The dynamic properties of the system may be inferred from the time sequence, or may be explicitly specified using an architectural composition language called the *evolution calculus*.

The calculus consists of a small number of operations allowing for the creation, destruction, connection, disconnection, synchronization and de-synchronization ("versioning") of workspace components and connectors, as well as their migration from workspace to workspace. The evolution calculus is a mathematically specified language that allows rigorous reasoning about architectures, such as whether a conceptual or implementation architecture is semantically sound, and whether an implementation architecture is a valid refinement of a given conceptual architecture.

3.2 Janet and Len Shop for a Car

We now present a simple collaborative scenario, which we model using the workspace conceptual-level notation. In subsequent sections we discuss how the conceptual model leads to component design and thence to a distributed system implementation.

Janet and Len are in the market for a new car and have narrowed their selection to three vehicle types from a single vendor. One Saturday morning, while Len is occupied with other business, Janet heads for her living room to begin shopping in

earnest. She places a call to the local car dealership and is connected with a salesman, David.

The conceptual-level model of the collaboration at this point is presented in figure 2. The dashed lines indicate workspace boundaries; the stick figures represent users; and the other items in the diagram represent components and connectors inhabiting the workspaces.

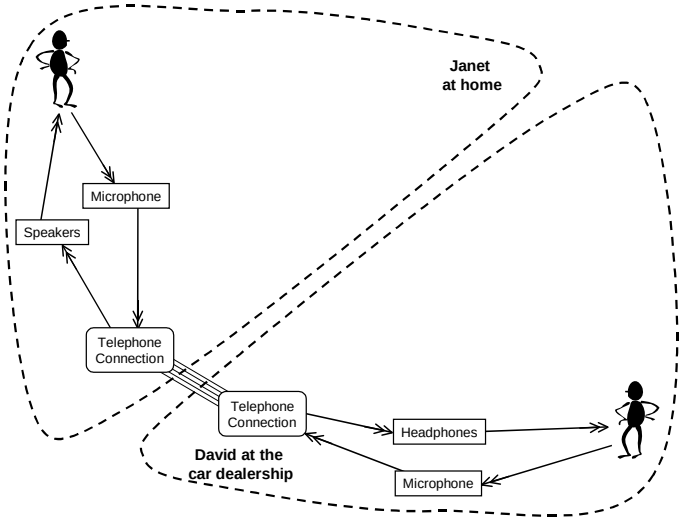

Fig. 2. Janet calls David.

At the conceptual level, users are modeled as a particular type of *actor* — a workspace element that is capable of initiating activity within the workspace. Non-human actors may be either hardware or software, and are represented by octagons.

All actors, including human users, may be the source and target of *subscription connectors*, which are indicated by double-headed arrows. Subscription connectors represent asynchronous (non-blocking send) channels by which events are delivered. Events may be information-rich objects like video frames, or simple indications like the update event in the classic Model-View-Controller pattern [8]. Subscription connectors may have multiple sources and multiple targets; in effect they are event buses, similar to those of C2 [14].

A subscription connector pointing towards a human user indicates that the user is paying attention to the connector's source. So, for example, the subscription arrow pointing towards Janet in figure 2 indicates that Janet is listening to the sounds produced by her speakers. Conversely, a subscription connector pointing from a user to a device indicates that the user is providing input to the device; in figure 1 the microphone is picking up Janet's speech.

Rectangles are used to represent hardware and software components that are passive and act only in response to external stimuli. Such components are called *reactors.* In figure 2, these are the hardware devices used in the telephone connection. A reactor may be the source and target of subscription connectors as well as call connectors. Call connectors, which appear in figures 3 and 4, represent blocking

method invocations, possibly with return values. Calls that modify their targets are referred to as *updates*; calls that return values are *requests*; and calls that do both are *request-updates*.

Rectangles with rounded corners represent *stores*, which are components containing shareable data. Stores are similar to reactors, except that they may not be the source of call connectors and are able to participate *synchronization groups*. A synchronization group is a group of stores that behave in a mutually consistent fashion, in effect as though they were a single store. In the Workspace Model, synchronizations are the only workspace constructs that are allowed to bridge between workspaces.

In figure 2, the synchronization notation between the telephone connections in Janet and David's workspaces indicates that these are conceptually "the same" telephone connection. More precisely, stores within a synchronization group are required to be consistent in two senses. Identical requests made of two stores in the group at the same time are required to return consistent values, and stores in the group are required to produce consistent event streams. The definition of "consistent" may be application-specific and may include a time component. For example the state of one store, and the event streams it produces, may lag those of another store by some period. The strongest form of consistency is strict observational equivalence. A fully specified scenario at the conceptual level will provide a definition of "consistent" for any included synchronization group.

Let us return to our scenario. In figure 3, Janet has described to David the vehicles that she is interested in, and David has offered to walk her through a video presentation of the features of each. Janet accepts, and the video appears on her television screen. David's image is superimposed on the video, allowing him to point directly to features of interest.

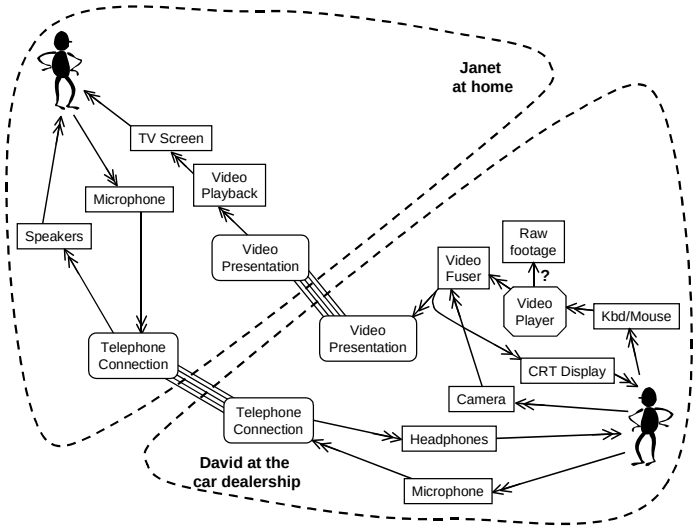

Fig. 3. David shows Janet a video presentation.

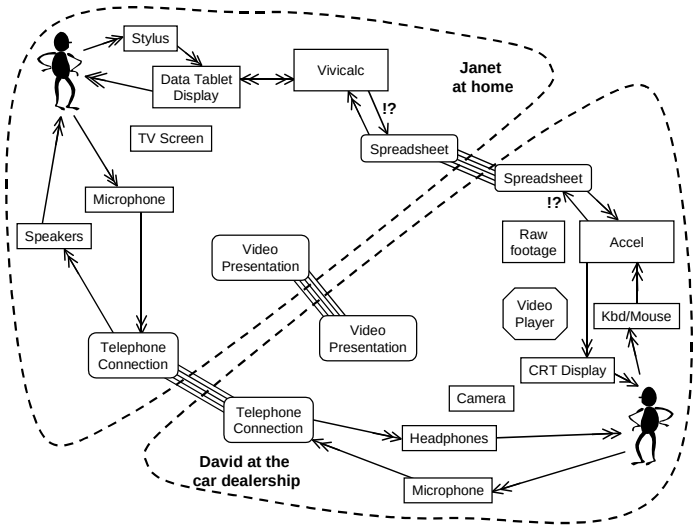

Fig. 4. Janet and David discuss price and financing**.**

While Janet and David are both looking at “the same” video presentation, they are using significantly different hardware and software. Consequently, their views of the presentation may be significantly different. Also, since Janet is a passive observer of the presentation, the subscription arrows in her workspace flow in one direction, while on David’s side there is considerably more complexity. Janet’s video playback component can be a simple reactor, passively displaying the frames provided to it by the video presentation; however, David’s video player must be an actor, since it is responsible for actively retrieving raw footage of the vehicles and “pushing” it to his video fuser.

After some discussion, Janet settles on a particular vehicle and begins negotiations regarding options, price, and financing. During the negotiations, David terminates the video presentation and provides a spreadsheet to help make the discussion concrete. In figure 4 we see Janet and David interacting with the spreadsheet using their preferred editing programs. Janet and David each have a call connector to the spreadsheet, which allows both requests, indicated by the question mark “**?**”, and updates, indicated by the exclamation point “**!**”. The subscription connector from spreadsheet to editor allows the editor to be notified of any changes that might have been made to the spreadsheet, and to update the display accordingly.

Eventually Janet and David come to a tentative agreement, subject to Len’s concurrence. Janet thanks David and makes the spreadsheet and video presentation persistent in her mobile workspace, which is hosted on her data tablet.

Later that day, Janet meets Len for lunch at a downtown restaurant. She brings her mobile workspace with her, and uses it to show him David’s video presentation and the spreadsheet they had worked out, as illustrated in figure 5. She accesses the spreadsheet exactly as before; however, she uses an active video playback component to retrieve and play portions of the stored presentation.

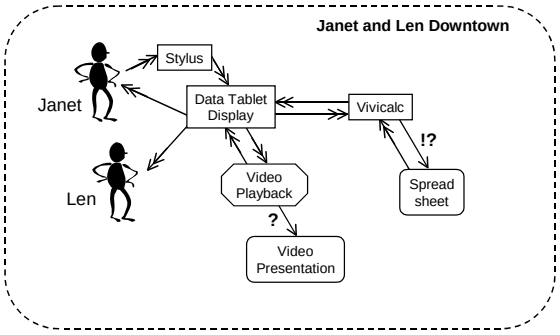

Fig. 5. Janet discusses the car purchase with Len.

Len suggests a few changes to the financing; Janet contacts David again and they agree on a delivery date.

4 From Scenario to Design and Code

Once one or more representative scenarios have been documented with workspace diagrams, we can proceed to system-level design and code. In this section we give a brief overview of these activities, highlighting design issues specific to the Workspace Model.

If the components and connectors in the workspace diagrams have been chosen carefully during scenario modeling, it will be possible to implement them directly in the Workspace Model. Ideally, each component will represent a well-defined functional entity and will be classified as a store, reactor or actor. Each element of separately shareable state will be in its own store. Each logically distinct, externally available capability of a component will have its own connector. Finally, all significant evolution operations affecting connectors and components will be identified. At this point we can proceed to design and code the components required to implement the scenario.

The attachment point of a connector to a component is represented as a *port*, either *source* or *target* depending on the required connector direction. A port will have a particular *vocabulary*, which is the set of calls or events that it originates or accepts. Source and target ports attached to the same connector must have compatible vocabularies. In effect, a target port's vocabulary is its interface and a source port's vocabulary is its type. The set of ports provided by a component determine how it can be connected to other components at run-time.

During execution, connectors may be either local or distributed, depending on the topology of the connected components. In support of the distributed case, call sources must be prepared for network failures. (Subscription connectors, which provide asynchronous event broadcast, never report delivery failure to the originating component.) In call source port definitions, update methods may be marked with a flag indicating that failure notifications are to be discarded. Similarly, request and request-update methods may have default return values, which are returned by the

run-time system if partial failure interrupts a request. Neither of these methods is sufficient to guarantee correctness in the event of failure; however, for user interface components they are often adequate. Methods that do not have default return values or an "ignore failure" indication will throw exceptions, which the component developer must be prepared to deal with.

5 From Design to Distributed System Deployment

Now that we have illustrated the conceptual level of the workspace model, and briefly discussed how the conceptual level guides component design, we turn our attention to the implementation of the collaboration as a distributed system.

At run-time, a series of conceptual-level evolution calculus operations will be submitted to the workspace run-time system for execution. Each of these represents an evolution of the workspace's configuration, and must be refined into a valid implementation. The general approach is to first anchor the components to particular host platforms or *nodes*, then to provide implementations for the components, and lastly to provide implementations for connectors and synchronization groups. Each time an evolution calculus operation alters the conceptual-level configuration of a workspace, the current implementation-level configuration is revisited to identify the ways in which it is no longer valid. The implementation level is then modified, also via the evolution calculus, to bring the conceptual and implementation levels back into alignment.

The implementation level of the workspace includes the developer-provided component implementations discussed in section 4, as well as the implementation-level connectors and run-time support components shown in figure 1. These last are provided as part of the workspace run-time system.

At the workspace conceptual level there is no explicit representation of the host computers that support workspace connectors or components. At the implementation level, host computers (or, more specifically, processes executing on behalf of workspace owners on those computers) are referred to as nodes, and are represented using the same notation as in Unified Modeling Language deployment diagrams [12].

The task of the workspace run-time system is to provide an implementation of the desired conceptual-level configuration of components and connectors on the available nodes, taking into account the available adapters and network connections, as well as any other factors of interest. Figure 6 illustrates the problem for the spreadsheet portion of our scenario: the run-time system must map the conceptual architecture of figure 6a onto the available workspace nodes shown in figure 6b. In figure 6 and the figures that follow we have elided the hardware components for simplicity, and we assume that some suitable network connects all nodes.

Consider first David's portion of the architecture. David has a spreadsheet store and an editor reactor in his workspace, and has desktop and server nodes available on which to implement them. The first task of the run-time system is to map conceptual components onto nodes. This mapping may be user-directed and manual, or it may be automatic and take into account any number of relevant factors, such as node and link capacity and performance, security requirements, persistence, availability, and so on

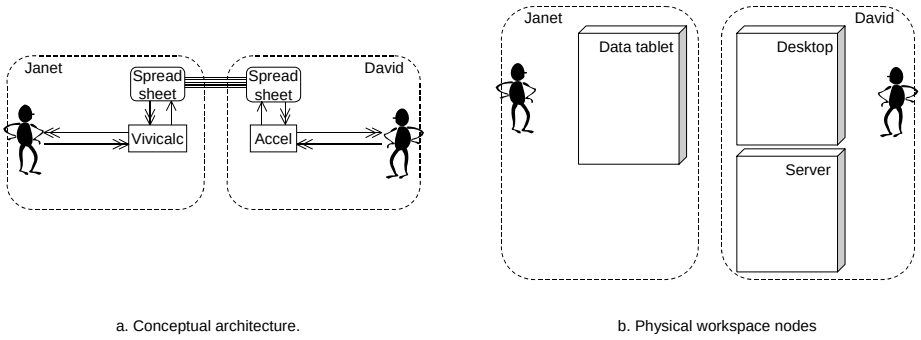

Fig. 6. Conceptual architectures must be mapped onto physical nodes.

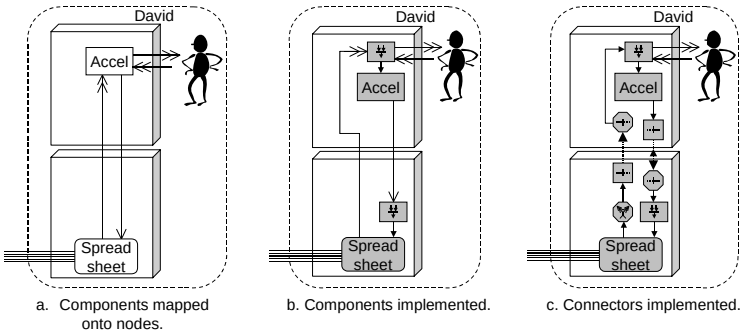

Fig. 7. Steps in implementing a simple architecture.

Initially, the conceptual components are considered to be "floating" in the workspace. The first task of the workspace run-time is to anchor the components to particular nodes, as illustrated in figure 7a. In this case the editor component has been mapped to David's desktop workstation, perhaps to simplify the connection from his input and display devices to the editor's user interface. The store representing the spreadsheet has been implemented on a server.

The next step is to provide implementations for the components themselves. Normally, these consist of two parts, as shown in figure 7b. One part is the component implementation itself, developed as discussed in section 4. The other part is a concurrency control and consistency maintenance component (CC/CM). As indicated earlier, component implementations may be written without regard to thread safety, since the workspace run-time guarantees that there will only be a single thread of execution within a component at any one time. This guarantee is provided by a CC/CM associated with each component implementation. All call and subscription connectors that target a component are routed through that component's CC/CM.

The final step in realizing David's portion of the workspace is the implementation of the connectors between his components. Since subscription connectors are asynchronous, active event broadcaster components are normally required for their

implementation. Event sources deliver their events to event broadcasters where they are immediately enqueued. The event broadcaster's internal thread then takes responsibility for delivery of the event to all subscription targets, allowing the event source to proceed with other computation. An event broadcaster is visible on the server node in figure 7c.

Where connectors cross process boundaries, interprocess communication (IPC) is required. This is provided by network transmitter and receiver components, which support message-based IPC. The message transmitter is a passive component, taking its thread of control from the component that calls it. Conversely, the receiver is effectively a server component, and therefore provides its own thread.[2]

Transmitter and receiver pairs support two messaging protocols. All conceptual-level calls are synchronous; therefore, a request-reply protocol is required for the implementation of distributed call connectors. However, since subscription connectors are inherently asynchronous, a request only protocol with robust delivery, as afforded by TCP, is sufficient. Transmitter and receiver pairs may be seen in the implementation of both the subscription and call connectors in figure 7c.

Temporarily ignoring the synchronization group, figure 7c represents a complete implementation-level architecture for David's spreadsheet and editor components and their connections. There are typically several valid refinements corresponding to any conceptual-level configuration, the knowledge of which is embedded in the workspace run-time system. In this example, the event broadcaster has been implemented on David's server node. It would be equally valid to implement it on the workstation node and to have the transmit/receive pair "upstream" from it; or to have an event broadcaster on each node; or to eliminate them entirely, since there is only one target and the request-only message protocol implemented by the transmit/receive pair provides the desired asynchronous semantics. Similarly, distributed call connectors may be implemented as shown here, or with the addition of cache and mirror cache components to eliminate latency for repeated invocations of the same request.

We now turn our attention to synchronization groups. The valid refinements for synchronization include both centralized and replicated implementations. In a centralized implementation, as illustrated in figure 8a, there is one copy of the shared component. In this implementation consistency maintenance is trivial; however, performance may suffer as a result of network latency. Figure 8a includes call and subscription connectors that cross workspace boundaries, apparently contravening the rule that only synchronizations may do so. However, since these connectors are part of an implementation-level diagram, rather than a conceptual diagram, the rule does not apply. Naturally these connectors would themselves require implementations.

Figure 8b illustrates a replicated implementation. To ensure that the replicas maintain the required degree of consistency, the CC/CM components attached to each of the replicas communicate with one another and enact a replica consistency maintenance protocol. For the two-replica case a bi-directional call would suffice as a

[2] Complex workspaces can rapidly accumulate a large number of network transmitter and receiver components. This is generally wasteful of operating system resources including sockets and threads. In practice, a single multiplexed transmit-receive pair in each direction can be used to implement a group of connectors between two nodes; thread pools can be used to provide responsiveness while reducing total thread overhead.

communications mechanism. However, for three or more replicas a group communication channel such as that provided by Spread [1] offers a more convenient abstraction. This approach is illustrated in figure 8b, where the CC/CM at each of the two replicas is connected to a multicast channel endpoint with channel identifier 1.

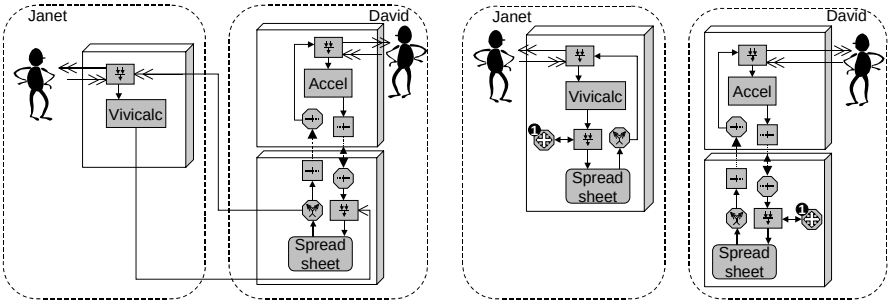

Fig. 8. Two possible implementations of a two-store synchronization group.

Any one of several replica consistency maintenance protocols may be used. These include the null protocol where updates are reliably broadcast but order is not enforced; optimistic protocols with rollback; locking; centralized or distributed strict ordering; and concurrent update protocols such as dOpt [13] or ORESTE [7]. The protocol implementations are provided in the runtime system as plug-replaceable sub-components of the CC/CM components. All but the concurrent update protocols can be implemented entirely by the runtime system. The concurrent update protocols require that the component implementer provide a protocol implementation including the necessary operational transforms or undo/redo operations. Currently, all replica maintenance sub-components in a synchronization group are required to be of the same type.

This concludes our overview of the normal operation of the workspace run-time system. We close the section with a few words about how the workspace handles distributed system failure.

Distributed system failures are detected by the local workspace run-time system using mechanisms provided by the underlying communications libraries (e.g., socket exceptions) and treated as evolution calculus operations. For example, the failure of a link supporting a call connector is treated as a destroy operation on that connector. If the connector's local end was its source, and an update or request was interrupted by the failure, the run-time system takes appropriate action as specified in the source port definition. The run-time system then implements the destroy operation by detaching the local end of the connector from its local source or target component, and by destroying any supporting components that exist only to implement the connector. In this way partial failure is handled as a normal occurrence in the system, rather than as an exceptional condition.

6 Status of Implementation

We are in the process of developing two toolkits and run-time systems supporting the Workspace Model. One, developed by Phillips and Graham, is being written in and for the Python programming language. The other, developed by Wolfe and Graham, is in C++. At the moment, both are capable of automatically providing run-time implementations of complex architectures on a single node. Distributed implementations are nearing completion.

The two implementations are not intended to be interoperable, although multi-language support within and between workspaces is a long-term goal. Rather, the aim is to see how best to integrate workspace constructs within these very different programming languages. In both cases, the toolkits allow simple "workspace-oblivious" components to be written idiomatically in the toolkit language and to be executed either in a stand-alone mode or within the workspace run-time. "Workspace-aware" components that take advantage of the workspace run-time services can also be written; obviously these require the workspace run-time to function.

The system in Python has made significant use of the language's dynamic and dynamically typed nature, as well as its metaprogramming interfaces. These have made the initial implementation of the workspace run-time system relatively painless. For example, the implementation requires just a single, generic event broadcaster class to implement any type of subscription connector. However, since Python does not have an inherent mechanism for interface specification, it has been necessary to develop a mini-language and conventions for the specification of ports and connector protocols.

In C++ the situation is reversed. Type-safety considerations in C++ complicate the development of workspace support components such as event broadcasters. In effect, a specialized event broadcaster is required for each type of subscription connector; the same is true for all other support component types. Wolfe's approach has been to allow the programmer to develop components in idiomatic and valid C++, and to use a pre-processor to extract port definitions and generate specialized type-safe variants of all required run-time support components. With the application of a few simple conventions, there is enough information in C++ method signatures to support the necessary Workspace constructs.

Acknowledgements

This work was partially supported by Communications and Information Technology Ontario (CITO), the Natural Science and Engineering Research Council (NSERC), and the European TACIT TMR Network. Nick Graham would like to thank Joelle Coutaz and the members of the IIHM lab for the opportunity to carry out the early stages of this work at the University of Grenoble, and Leon Watts for many stimulating discussions of these ideas.

References

1. Y. Amir, C. Danilov, and J. Stanton. A low latency, loss tolerant architectureand protocol for wide area group communication. In *Proceedings of the International Conference on Dependable Systems and Networks* (FTCS-30, DCCA-8, New York, NY), June 2000. Also available from `www.spread.org`.
2. G.E. Anderson, T.C.N. Graham, and T.N. Wright. Dragonfly: Linking conceptual and implementation architectures of multiuser interactive systems. In *Proceedings of the 22nd International Conference on Software Engineering* (ICSE '00, Limerick, Ireland, June 4–9), 2000.
3. R.M. Baecker. *Readings in Groupware and Computer-Supported Cooperative Work: Assisting Human-Human Collaboration*. Morgan Kaufmann Publishers, 1993. ISBN 1-55860-241-0.
4. A. Dix, D. Ramduny, and J. Wilkinson. Interaction in the large. *Interacting with Computers*, 11(1):9–32, December 1998.
5. E. Dubois, L. Nigay, and J. Troccaz. Consistency in augmented reality systems. In *Proceedings of the IFIP 2.7 Working Conference on Engineering for Human-Computer Interaction* (EHCI '01, Toronto, Canada, May), Published as Lecture Notes in Computer Science vol. 2254, pages 117–130. Springer-Verlag, 2001.
6. C.A. Ellis, S.J. Gibbs, and G.L. Rein. Groupware: Some issues and experiences. *Communications of the ACM* (also in [3]), 34(1):38–58, January 1991.
7. A. Karsenty and M. Beaudouin-Lafon. An algorithm for distributed groupware applications. In *Proceedings of the 13th International Conference on Distributed Computing Systems* (ICDCS), pages 195–202, 1993.
8. G.E. Krasner and S.T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object- Oriented Programming*, 1(3):26–49, August/September 1988.
9. J.C.R. Licklider. The computer as a communication device. *Science and Technology*, April 1968. Reprinted in Digital Systems Research Center Tech Note 61, August 7, 1990.
10. F. Paternò. *Model-based Design and Evaluation of Interactive Applications*. Springer-Verlag, November 1999. ISBN: 1-85233-155-0.
11. W.G. Phillips. Architectures for synchronous groupware. Technical Report 1999-425, Queen's University, Kingston, Ontario, Canada, May 1999. Available from `www.cs.queensu.ca`.
12. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1998. ISBN 0-201-30998-X.
13. C. Sun and C. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievments. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '98, Seattle, WA, USA), pages 59–68. ACM Press, 1998.
14. 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 Transactions on Software Engineering*, 22(6):390–406, June 1996.
15. T. Urnes and T.C.N. Graham. Flexibly mapping synchronous groupware architectures to distributed implementations. In *Proceedings of the Sixth Eurographics Workshop on Design, Specification and Verification of Interactive Systems* (DSV-IS '99), pages 133–148, 1999.
16. T. Urnes and R. Nejabi. Tools for implementing groupware: Survey and evaluation. Technical Report CS-94-03, York University, Canada, May 1994.
17. G.C. van der Veer and M. vanWelie. Task based groupware design: putting theory into practice. In D. Boyarski and W.A. Kellog, editors, *Proceedings of the ACM Conference on Designing Interactive Systems* (DIS '00, New York, USA, Aug. 17–91), pages 326–337. ACM Press, 2000.