Developing Collaborative Applications using Sliverware

Seth Holloway
Christine Julien

TR-UTEDGE-2006-005

Developing Collaborative Applications using Sliverware

Seth Holloway and Christine Julien

Mobile and Pervasive Computing Group
The Center for Excellence in Distributed Global Engironments
The University of Texas at Austin
{sethh, c.julien}@mail.utexas.edu,
http://mpc.ece.utexas.edu

Abstract. Despite computers' widespread use for personal applications, very few programming frameworks exist for creating synchronous collaborative applications. Existing research in CSCW (computer supported cooperative work), specifically approaches that attempt to make current application implementations collaboration-aware, are difficult to implement for two reasons: the systems are focused too narrowly (e.g., on Internet-only applications), or the systems are simply too complicated to be adopted (e.g., they are hard to set up and adapt to concrete applications). Enabling real-time collaboration demands lightweight, modular middleware—*sliverware*—that enables the fine-grained interactions required by collaborative applications. In this paper, we introduce sliverware and give a specific example in the guise of a *distributed keyboard* that multiplexes input from several users into a single stream that each user receives just like input from a normal keyboard. The result is simple, real-time collaboration based on a shared, distributed view of data that enables rapid development of highly coupled coordinating applications.

1 Introduction

Even though computers worldwide are connected, the computer is still largely a personal experience; CSCW (computer supported cooperative work) research has focused on expanding the limitations of traditional computers to allow for more collaboration amongst users. However, CSCW has mainly provided *information exchange* rather than *information sharing* [6]. While both allow for collaboration, information sharing provides real-time interaction with a consistent global view of shared data. Despite the large volume of work that has been done in connecting people, simultaneous collaboration and information sharing are still relegated to face-to-face meetings and telephone or video conferences. Existing research into enabling collaboration is too narrowly focused to be usable by everyone; for example, available solutions may only work online or with a specific operating system, or they may rely on a particular programming framework. Additionally, the approaches often rely on heavyweight, unchangeable middleware systems which provide excessive functionality; the vast amount of functions cannot be easily understood and fully utilized yet they dominate the software size.

This paper introduces *sliverware*, which aims to simplify the development of collaborative software. A sliverware component (a *sliver*) is a thin slice of a modular middleware that provides a specific functionality appropriate to implementing collaborative software. Each piece of sliverware includes not only the programming abstractions a developer uses to create the particular interaction, but also the protocol implementations required to realize that interaction. By hiding the details of this implementation from the application programmer, sliverware allows developers of collaborative software to focus on *interactions* instead of low-level communication. An application requiring multiple modes of interaction pulls together exactly the required sliverware components in a modular fashion to create a tailored middleware layer that implements the interactions' underlying behavior. By providing full application stacks within one sliver, collaborative programs can be constructed out of sliverware with very little knowledge of distributed programming. Common components among slivers are automatically optimized at compile time to provide the smallest-possible implementation. Traditional middleware provides a horizontal abstraction, but sliverware introduces the notion of lightweight, cross-cutting, vertical abstractions.

This paper details the sliverware concept including an overview of the approach, details of each abstraction layer, and a description of how to program using sliverware. We then show an example to demonstrate sliverware's flexibility. This example, the distributed keyboard, logically multiplexes multiple users' inputs into one input stream. By replacing the normal keyboard listener with a distributed keyboard listener, an application developer can program real-time text-based collaboration. There are numerous applications in which the distributed keyboard can play a critical role, including interactive presentations—lectures in which the professor's material is broadcast to students instantly while students create annotations on the fly—and simultaneous text editors that allow multiple people to author a document simultaneously. These possibilities make the abstraction ideally suited for educational software and software development tools although many more potential applications exist. Results show that the approach is easy to implement and scalable.

In Section 2 we motivate this research and define the problem that sliverware solves; Section 3 details sliverware and its approach to program construction. Section 4 covers the implementation details for one example of sliverware—the distributed keyboard, and in Section 5 we discuss work related to sliverware and collaborative software. Finally, we conclude with Section 6.

2 Motivation and Problem Definition

Real-time collaboration is long overdue in software; for nearly 40 years people have only had access to single-user applications. With the advent of Internet-based communication, many people saw the power of multi-user applications. Today, almost everyone communicates via a chat program such as AOL Instant Messenger [3]. However, while everyone is chatting with peers, they are work-

ing on single-user applications [13]. For increased collaboration, group members often meet face-to-face rather than using their computers directly. This is due largely to the fact that existing collaborative software provides only coarse-grained interactive capabilities. The types of tasks we commonly cooperate on demand varying levels of granularity and sharing of dynamic data in real time.

If such flexible interaction primitives are available, software could further enhance collaboration. For example, real-time collaboration in a shared word processor may improve productivity by allowing multiple authors to contribute simultaneously: everyone editing the document would see exactly the same up-to-date version. In a setting where collaboration is employed, contributors have a shared goal and work towards producing higher quality work in less time than is possible using only non-collaborative word processors. Another example is a collaborative software development environment that utilizes simultaneous text editing similar to the collaborative word processor. Agile development methods such as extreme programming (XP) [5] very nearly demand collaborative software environments. Rather than waiting for code to be checked into a version control system, programmers could view one another's progress in real time. The synchronous development environment may be an ideal collaboration target because the developer is also the end user, thus the end user has a solid understanding of the capabilities of software which avoids many of the traditional groupware pitfalls [8].

There is also great promise in enabling interactive lectures, particularly in an educational or industrial meeting. A growing number of lectures currently utilize computers through standard methods like slide shows and more diverse means such as electronic whiteboards [1]. Many of these technological improvements have removed several of the interactive qualities of traditional classroom environments. Enabling digital collaboration during lectures would foster an interactive environment where students personally augment the lecture with relevant annotations in real time. This method of note-taking gives students the opportunity to absorb the lecture and clarify notes without having to focus solely on copying the lecturer's words. The synchronicity also allows the audience to review the lecture and understand how different parts relate as opposed to seeing notes scattered in the margin. Real-time feedback from students enables lecturers to dynamically adapt the lecture's content and pace. These combined effects would allow for a truly interactive lecture with a room of active participants.

There are many CSCW ideas and applications related to this research, but none have received widespread usage due to several fundamental problems. First many collaborative approaches are complex and hard to use—the basic idea is incomprehensible or the development method is ill-defined. In addition, there is a general lack of programming tools tailored to enabling development of collaborative applications—the granularity of programming constructs is too low-level, focused on communication routines, and not tailored to the high-level interactive qualities of collaborative applications. The CSCW approaches are also too specific so they do not integrate with other systems easily. These combined challenges scare even seasoned programmers away.

In summary, existing programming constructs do not adequately address collaborative software developers' needs, including: expressive coordination and group membership primitives, an extensible and easy to use programming framework, and responsive communication-based interactions.

The work undertaken in this paper addresses the above challenges and presents the following specific essential characteristics:

- *Tailored collaborative abstractions*: we introduce abstractions that make collaborative programs easier to write and deploy quickly by encapsulating interactive capabilities within reusable programming constructs.
- *Extensible programming framework*: our approach allows the system to be modified by collaborative application developers at various levels of complexity.
- *Lightweight software footprint*: we provide a simple, efficient solution that dynamically minimizes the resources required by applications.

The next section introduces sliverware, which stands to overcome barriers to enabling collaboration in numerous ways. Sliverware supplies easy-to-understand abstractions tailored to collaboration functions so application developers can construct applications from existing components rather than having to program applications from the bottom up. This also frees the programmer from writing complicated, low-level communication methods. Finally, sliverware can be extended and used in three distinct ways, providing tailorable abstractions for programmers of all skill levels.

3 Sliverware: A Constructive Programming Model

Sliverware is thin modular middleware that offers the functionality necessary for writing collaborative software. Application developers do not need to know vast amounts of distributed computing solutions before using sliverware; instead the programming model provides a hierarchical collection of components that each implement a specific functionality necessary to collaborative software. The sliverware approach abstracts all aspects of collaborative programming into three layers: collaborative services (the collaborative functionality and application programming interface, or API), group membership, and network communication. A specific sliver teams three components (one from each layer) to provide an abstraction that allows programmers to develop collaborative applications quickly and effectively. Novice programmers can use particular slivers through the slivers' high-level programming interfaces, while more advanced developers can combine components in a different manner to create new slivers.

Sliverware provides all the functionality application developers have come to expect from middleware while also allowing for execution on more resource-constrained devices. The latter benefit stems from the fact that sliverware's modularity enables an application developer to incorporate only the middleware components required for the particular application instance. The framework implements many common collaboration tasks through the reuse of components.

The model is flexible because it can provide many varying potential solutions from a vast library and ultimately trims the system at compile time. Sliverware is also specifically collaborative-oriented which allows it to tailor the programming interfaces and functionality underlying them to a specific set of tasks. This simplifies the complexity of the development task while still enabling a large number of similar applications as described in the previous section. Finally, sliverware is component-based which provides developers the ability to mix and match components at varying levels of abstraction.

Fig. 1 depicts the sliverware-program interaction model and shows that sliverware interacts with four layers in the application stack: the network, group, service, and API levels. Fig. 2, the sliverware component model, shows the essential components of a single sliver. High level sliverware functionality is decomposed into pieces for each of the three layers. An underlying network protocol is necessary to enable coordinating partners to exchange information; this is handled by the network communication layer. It plugs into the sliver via an *adapter* that provides the communication interface, described in Section 3.3. Collaboration also requires the coordinating partners to be organized into a group that contains all of the distributed application components that must be kept consistent. The group membership layer provides support for organizing and coordinating the group itself, and may provide specialized logic for distributing information to all group members. This group membership and the group interface for connecting group policies to collaborative services is described in Section 3.2. Finally, we think of the implementation and API layers as a single entity that encapsulates the collaborative functionality which is separate from the coordination or distribution method. This layer, described in Section 3.1, incorporates algorithms that provide collaborative support for the local application such as the methods to allow information sharing (e.g., creating periodic snapshots to build a consistent global view or processing snapshots from other group members). In the remainder of this section, we describe the responsibilities of each of these component layers in more detail. We then describe how developers program

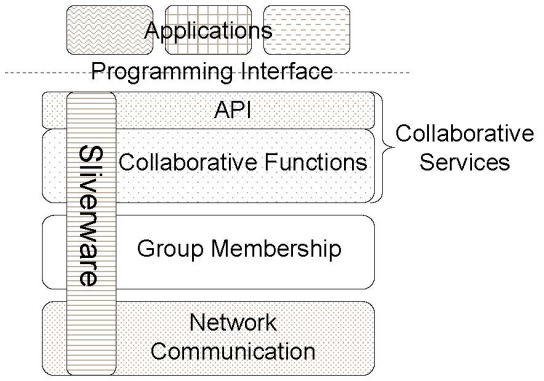

Fig. 1. Sliverware-program interaction model. Sliverware extends vertically through three layers: the network communication, group membership, and cooperative services layers. The sliverware hooks into the application via an API.

with slivers and how slivers can be combined automatically to optimize the resultant, tailored sliverware.

3.1 Collaborative Services

The uppermost layer in the sliverware model logically combines the collaborative functions and an API that presents the programming model to the developer to create a suite of *collaborative services*. This layer is founded on the premise that programming collaborative applications can be made easier if the programming primitives focus on functions integral to collaboration while hiding the necessary underlying group cooperation and network communication protocols. Implementations at this layer provide discrete fine-grained services that enable different pieces of functionality required for collaborative applications. For example, collaborative applications may exchange input information (e.g., keyboard or mouse events) occurring across a set of distributed devices, diffs of the same file open concurrently on multiple machines, simple idle/active status information of participants, or even screen shots for each connected user. For each such function, a collaborative service is defined that implements exactly the behavior dictated. A collaborative application can then be defined by integrating one or more such collaborative functions and providing application-specific behavior on top. This application-specific behavior is not part of the sliverware model, and includes determining how shared information is displayed and how often information should be exchanged. Some collaborative services may also automatically collect context information from the user or device generating data and attach it as part of the collaboration information.

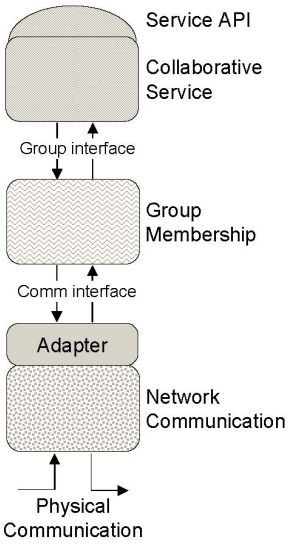

Fig. 2. Sliverware component model. An individual sliver combines one component from each of the three layers.

Section 4 gives an example of a specific collaborative function, the distributed keyboard. In this collaborative service, every participant in the collaboration group has attached a keyboard listener to a component in the application. When a keyboard event occurs in that component, the event is not only displayed appropriately on the user's local device (e.g., by displaying the letter typed), but the event is also automatically distributed to all other group participants. The distributed keyboard's API is defined to interact with the application in both directions (event generation and event reception). The information exchanged when interacting with different collaborative services likely differs greatly (e.g., interacting with a distributed keyboard listener is clearly quite different from interacting with a distributed mouse listener). In the sliverware model, we present

this service interaction through a common `serviceListener` that can be subclassed by a sliver component developer to be made to resemble local interfaces with which the developer is familiar. All context collection and event distribution is hidden from the application programmer and handled by the implementation of the collaborative service, and, ultimately, by the underlying communication and coordination protocols described next.

3.2 Group Membership

One of the most important aspects to supporting the collaborative services described above is the ability to support collaborative groups. Without defining the group that is collaborating (even if this group contains only two participants), then it is difficult to understand and specify the collaborative behavior (in the layer above) or to implement the collaborative behavior (which relies on the communication protocols described in the next section). For these reasons, group membership specification and implementation is the second, middle component, of any sliver in our model, and this component handles all steps necessary to store, collect, and learn group identities. The group membership policy that dictates how members are added to the group can present varying levels of complexity depending on how restrictive, consistent, and/or private the group should be. For example, the simplest group policy allows anyone who hears a communication to be part of that group. This policy offers no consistency guarantees however, because there is no information provided about which parties have received the message. More complicated group policies may require potential members to explicitly join, they may require members to periodically update their status to remain active, or they may even require members to know a shared password to participate.

Regardless of their semantics, groups are represented by well-defined data structures that contain the following components, if applicable:

$$group \triangleq (groupName,\ groupAddress,\ members)$$

This group data structure is maintained independently by each member; unless explicitly implemented by the group policy, no guarantee is provided that every group member has the same view of the current group. The *groupName* is a unique group identifier, *groupAddress* contains the group IP address for multicast purposes, and finally *members* is a list of all group participants. Any of these fields can be empty; though such null values make some sliver combinations impossible (for example, if the multicast address is not set, but the underlying communication is provided by a multicast protocol, there is no way to contact the other group members). These subtleties, their implications, and how our sliverware model and framework handles them are discussed in the next section. The *members* portion of the *group* data structure is a list of individual members each presented in the form:

$$members \triangleq (name,\ address,\ attributes)$$

where *name* is the group member's identifier, *address* is the group member's IP address, and *attributes* contain any other information that the program may track. As with the *group* data structure, none of these aspects are required; if the underlying communication protocol is using a multicast address, it is possible that an IP address for each individual member is superfluous. The *attributes* are largely open-ended, and different group policies may use them for different purposes. As one example, if the group policy requires periodic *heartbeats* from each group member, the time to the next expected heartbeat would be included as an attribute for each member. Context information about the node, such as physical location, can also be included in the attributes.

The actual implementation of a group membership component depends on the particular policy employed by the group membership protocol. In our prototype, we present two widely applicable group membership policies: `announce` and `n-heartbeat`. The first of these, `announce`, provides open access to the group that allows anyone to join by simply announcing their intent. The policy does not attempt to prune unproductive members and instead relies on explicit departures from the group. To join the group, an application component's group membership component broadcasts a `join` request for the group. This is received by every node in the network, and, if the node supports an application participating in the specified group, the associated group data structure is updated with information about the new member. At this point, when any member of the group sends collaborative information, it is received by all other members specified in the group data structure *if they are still connected*. When an application component wishes to depart the group, it sends a `depart` message to all nodes it knows to be part of the group.

While the `announce` policy is useful for supporting many applications that are relatively static, more complex policies are necessary to support more sophisticated applications. For example, more stringent joining processes or more rigorous exit criteria may be programmed to suit the application. This brings us to our second prototype group membership policy, `n-heartbeat`, which requires that all participants periodically broadcast heartbeat messages; if a member does not communicate within `n` heartbeat periods, the member is assumed to have disappeared and is removed from each member's group data structure. Each member performs this removal independently; there is no guarantee that all members will have the exact same lists at all times, but if a node is no longer sending heartbeat messages, every active group member will eventually remove that node from the group data structure. To join a group, a new member sends a `join` request as above, but in addition, the node must periodically send a short message indicating it is still present and active. Still more sophisticated group membership policies are possible, for example, based on relative physical location and its implied notion of future connectivity [9].

Because of its modular design, the sliverware framework allows for more group membership policies to be integrated quickly and easily. A group membership component developer simply needs to provide functionality for joining the group and any constraints on the group interactions. The interface the

group membership layer presents to the collaborative service layer contains a `send(Message m)` method (for calls coming from a collaborative service above) and a `GroupListener` for delivering group events from other group members to the collaborative service. Before invoking the collaborative service's listener, the group membership implementation is responsible for ensuring that the message received is in fact destined for this group and that the sender is a part of the group. The interface the group membership protocols use to interact with the lower, network communication layer contains some additional methods and is discussed in more detail next. In summary, to add a new group policy to the framework, the policy developer is responsible for understanding how to use the underlying communication interface, for providing an implementation of the `send` method called by the collaborative service, and for invoking any registered group listener to deliver messages to a registered collaborative service.

3.3 Network Communication for Collaboration

Communication is a necessary part of collaboration—without proper communication there is no collaboration. With real-time collaboration in mind, communication should generally include every group member, be immediate, and ensure that every contribution that is made is registered. However, many factors influence the quality and cost of communication, including the degree of synchronicity, priority requirements, timeliness, etc. Combinations of these factors lead to numerous ways to provide communication for collaborative applications. Complicating this is the fact that different applications have different performance requirements and communication approaches perform differently when faced with different operating constraints. Given these discrepancies, it becomes apparent that there is no common communication approach that is the best choice for all applications or even for a particular application in every scenario. For this reason, the sliverware programming model simply defines a common interface, or *adapter* [7], that communication approaches must adhere to in order to be usable by the other, higher level, sliverware components. This is similar to the requirement above that new group policies adhere to the specified interface, but in this case we make the particular interface an explicit individual component because we do not modify the communication protocols themselves. Because a standard interface is used, an application developer can swap in different communication implementations without altering any of the higher level implementation components.

The basic interface for collaborative communication in our sliverware model allows both communication with a single member of the collaborative group and simultaneous communication with the entire group. While reliable multicast [12] appears at first glance to be an obvious widely applicable solution, it is difficult to implement in the dynamic, lightweight, wireless systems we are targeting [11]. A reliable multicast implementation may be the correct choice in some cases, but, depending on the situation, a simpler flooding-based broadcast may be sufficient. In other cases, more tailored protocols may be appropriate. For example a cooperative text editor that will only involve a select group of individuals in the

same boardroom (such as in a meeting) could utilize simple flooding, while the same application connecting a small group within in a large audience (such as a conference) may use ad-hoc on-demand distance vector routing (AODV) [15] or multicast ad-hoc on-demand distance vector routing (MAODV) [16]. In either case, the application developer needs only to select the proper network communication component for the sliver and the common send interface (defined by the adapter) will distribute the necessary data to the group.

Method	**Description**
`join(String name)`	called by the group membership layer on the network communication layer to enter the callee into the group with the specified name
`depart(String name)`	called by the group membership layer on the network communication layer to remove the callee from the specified group
`send(Address a, Message m)`	called by the group membership layer on the network communication layer to send the specified message to a single recipient: the node indicated by the specified address
`sendAll(Group g, Message m)`	called by the group membership layer on the network communication layer to send the specified message to all members of the specified group
`receive(Message m)`	called by the network communication layer on the group membership layer to deliver the specified message; implemented within the `CommunicationListener`

Fig. 3. The interface between group membership and network communication

The methods comprising the interface between group membership and network communication are shown in Fig. 3. The `join` and `depart` methods were discussed previously; they are used by some group membership protocols that need to explicitly join and leave groups. At first glance, one might think these methods ought to be implemented within the group membership layer itself and not involve the network communication layer, but some join activities require participation of the communication protocol (e.g., joining a multicast group) while others do not (e.g., communication that relies on broadcast for every message). Because our foremost goal is pushing all knowledge of communication to the network layer, we require the group membership layer to know only the name of a group in which it participates, and the protocol delegates knowledge of how the communication protocol operates to whatever implementation resides in the network communication layer of a particular sliver. In this manner, the group membership protocol performs exactly the same behavior regardless of the nature of the underlying communication protocol.

The remaining three methods allow data to be exchanged among group members. The common element in these three methods, `Message`, includes the *group name*, the *source*, and the *data* to be transmitted. The functionality required to send a message to everyone in the group changes with the networking protocol, but the interface distances the application developer from low level details. Overall multicast functionality is provided by the `sendAll(group, message)` method; here, *group* refers to the data structure presented earlier that contains the *group name*, *group address*, and *members* list. Recall, however, that the *group* data structure does not have to contain each of these elements for the communication protocol to be able to function correctly. For example, AODV does not support multicast, so the *group address* provides no useful information. However, if the elements of the *members* list are omitted, AODV cannot reach the other group members since it relies on unicasting the message to each of them. In this case, the method triggers a `NoMulticast` exception. On the other hand, MAODV is a multicast protocol that automatically provides group-wide communication. In this case we expect the *group* element to contain a *group address*, and we trigger an exception if the element is undefined. It is important to note that we do not alter the communication protocol implementations themselves and instead implement these behaviors in the adapter interface. As an example, the code implementing `sendAll` for an AODV adapter looks like the following:

```
void sendAll(Group g, Message m) throws NoMulticast, UnreachableHost {
   if(g.members == null){
      throw new NoMulticast();
      return;
   }
   for(int index = 0, index < g.members.size(); index++){
      if(g.members[index].address != null){
         AODVSend(g.members[index].address, m);
      }
      else{
         throw new UnreachableHost(g.members[index]);
      }
   }
}
```

The above is an example of the simple interfacing code that a sliver developer must write to be able to include a new communication protocol as a component in the network communication layer.

If the network communication is implemented by a brute-force broadcast protocol, the `sendAll` method will deliver the message to every connected node, even those that do not support members of this particular group. As the group membership layer is communication agnostic, so the network communication layer is group agnostic. When the network communication layer receives a message, it delivers it to the group membership layer. It is then the group membership protocol's responsibility to filter these messages and ensure only proper messages are delivered to the collaborative service and ultimately the application.

There may also be a need for communication between single members of the group; this may be used to implement `join` and `depart`, but it may also be useful for sidebar coordination within the group. This unicast style of behavior is achieved through the `send` method in Fig. 3. In AODV this method is the base functionality, so little extra work is necessary. In the case of MAODV and flooding we rely on the group membership layer at the receiving end to filter all messages where the target address does not match the host address.

All incoming messages trigger the `receive` method in the `CommunicationListener` interface. This method's implementation is likely quite simple in all cases; it is a basic pass-through of information from the network to the group membership layer. If the message received was not sent by a member of the group, or if the group identifier in the message does not match a group id of the receiver, the group membership layer filters the information rather than passing it to the application layer.

Our prototype contains an implementation for flooding, unicast, and multicast based protocols; a sliver component programmer can easily insert another protocol by creating an adapter for that protocol. In no cases is it necessary for the programmer to modify the existing communication protocol implementation; the adapter simply changes the interface to conform to our framework. Therefore, this system allows very low-impact changes to the network layer while providing extreme flexibility and extensibility.

3.4 Programming with Sliverware

Sliverware is an extensible, modular model for enabling collaboration in applications. The abstractions provide collaborative services that are easy to understand and use. Referring back to Fig. 2, each sliver provides a discrete collaborative function and comprises the local implementation of that function, the implementation of a group membership policy that defines who participates in that collaboration, and a network communication component that facilitates physical message exchange. Programming collaborative applications with the sliverware model is straightforward and simply requires a developer to select, use, and combine slivers that provide the high-level collaborative functions the application demands. This programming may result in a sophisticated application that encompasses several collaborative functions and may even combine views of information from multiple different groups. For example, in a classroom support application, a group may be defined for the entire class so the teacher can share materials with every student, and there may be additional groups defined for a team project. A single student's display may simultaneously show information from both of these collaborations side by side.

From the programmer's perspective, each of these behaviors is programmed independently as a single sliver. When the application is compiled, the sliverware framework optimizes the resulting implementation by combining functionality that is duplicated across slivers. Fig. 4 shows this process in a bit of detail. The left of the figure pictorially represents an application that combines three separate slivers: a distributed keyboard shared with all of the members of the class,

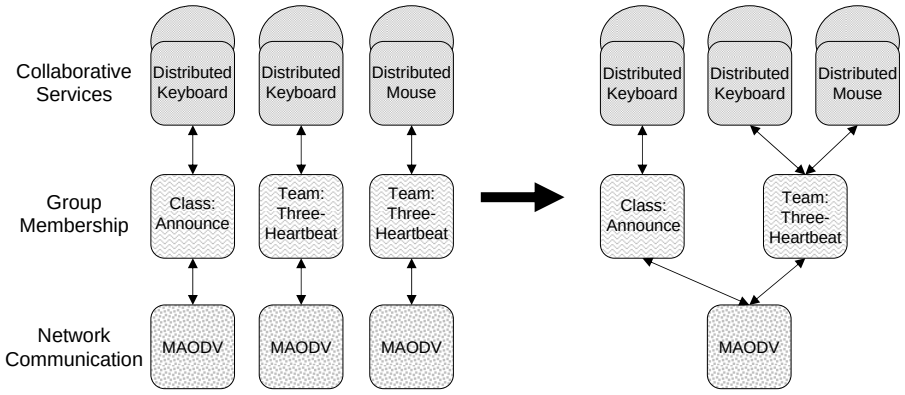

Fig. 4. A visualization of the sliverware simplification; components that are reused are combined to create a minimal set of functionality that is lightweight yet functional.

a distributed keyboard shared with the team members, and a distributed mouse shared with the team members. The classroom application may have a dedicated window for keyboard events shared with the class and a separate window for keyboard and mouse events shared with the team. When a keyboard event occurs, the collaborative application implementation (not shown) first determines which window had the focus and then triggers the appropriate collaborative service.

When compiled, the actual middleware deployed in support of the application is shown on the right of Fig. 4. The duplicate components (i.e., the MAODV implementation used and the group membership policy implementation for the team) have been combined for efficiency. When a message is received at a node, the MAODV implementation passes it to both group implementations (because it doesn't know how to read or process group information). Each group processes only the messages destined for its members (as indicated by the *groupName* carried in the message). The team group implementation passes all group messages to both collaborative services (i.e., both the distributed keyboard and the distributed mouse), but each of these services only passes along events whose data portions match the type for that service. In general, duplicated network communication protocols are always combined. Ultimately, the compiler optimizes the set of slivers to create a tailored lightweight middleware. This is in contrast to traditional approaches where an entire middleware system must be deployed and invariably contains functionality that will not be employed by *every* application. Sliverware's approach leads to a much leaner execution environment which can translate into increased performance and greater support for heterogeneity.

The flexibility described above illustrates an important property of the sliverware programming model: the ability to create new slivers by adding and exchanging components. Overall, sliverware can be programmed by three distinct classes of developers with increasing levels of programming expertise. First are the collaborative application developers who access suites of available preconfig-

ured slivers and combine them into applications as described above. Second are the sliver developers who combine sliverware components from each of the three layers to create new combinations presented as slivers. Finally, the most experienced class of programmers, sliverware component programmers can develop new sliverware components that can be used in sliver combinations. For each component level, we discussed previously the steps this developer must take to create a new implementation that adheres to the well-defined sliverware model.

Sliverware allows users of all skill levels to use and extend the sliverware system by the unique hierarchical structure of the sliverware approach. Multiple layers of abstraction lead from the low-level implementation to useful high-level functionality. By providing a complete framework based on multiple layers of abstraction, sliverware is a flexible, extensible, and reusable model that provides collaborative services.

4 An Example Sliver: the Distributed Keyboard

Sliverware enables collaborative applications to be developed quickly and removes repetitive, burdensome communication and coordination methods. To aid understanding of sliverware we present an example application, a chat program in which a group of users shares keyboard inputs that occur within the application. The collaboration is achieved using a distributed input device, the distributed keyboard. The distributed keyboard listener replaces the traditional keyboard listener and multiplexes all the users' key-presses from within the application into a stream similar to the input from a standard keyboard.

Fig. 5(a) demonstrates how the traditional keyboard listener connects individual users to their machines. Fig. 5(b) shows the simple approach taken by the distributed keyboard listener; the individual inputs are joined into one logical distributed input device which is fed to each machine. This approach allows individuals to remain at their own computer while maintaining a consistent global view. The distributed keyboard appears to function as a standard client-server implementation, however, instead of routing all packets through a server, the sliver automatically multicasts users' input to the group. Bypassing a central server speeds the implementation, allowing for more immediate interactions. The result is a chat program that immediately broadcasts each group member's key-preses and displays the input in a text box which is consistent across the group.

Fig. 6 shows the application stack underlying the Chat Application implementation. The collaborative application programmer contributes only the upper-most block of this stack. The programmer uses the `ServiceListener` interface and the `notify` method to plug into the sliver used (in this case, a distributed keyboard implementation). The distributed keyboard demultiplexes events from other group members and delivers them to the application. It also receives the local user's events, packages them, and sends them to the group implementation (in this case, an instance of the `announce` group). Through the `GroupListener` interface, the `announce` group implementation distributes

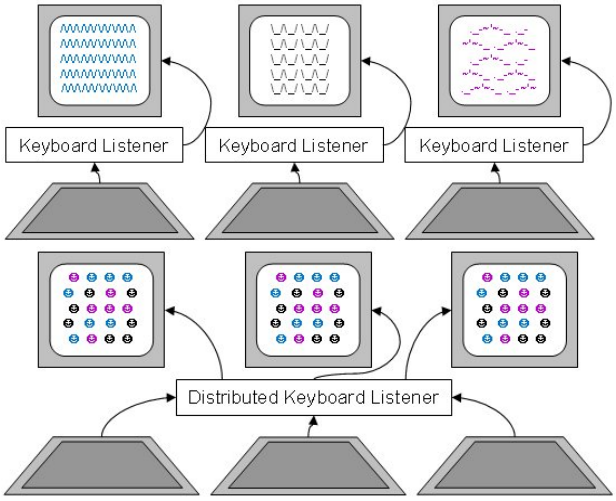

Fig. 5. The traditional keyboard listener connects a single user to a computer (top). The distributed keyboard listener connects multiple users while still allowing them to use their local machine (bottom).

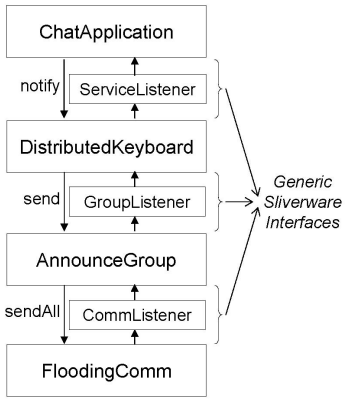

Fig. 6. The application stack for our Chat Application.

incoming events to the registered collaborative service (i.e., the distributed keyboard). The `announce` group implementation also passes the message from the local collaborative service on to the communication implementation through the `sendAll` method which delivers the key event to all other group members. The interfaces between the layers in Fig. 6 are generic sliverware interfaces included in the sliverware framework. Regardless of the type of component employed at each layer, it is a given that the same interfaces are used. These generic interfaces enable sliverware components to be exchanged for each other without impacting other components in the sliver.

Fig. 7 shows a screen shot the sliverware chat application; the implementation can be found on our group webpage at `http://mpc.ece.utexas.edu/research/sliverware.html`.. Each group member appears in the chat window; as the user types, their key-presses are displayed in real-time on all connected users' screens. This particular application chooses to display distributed key events from different users based on the events' relative times of arrival. Other application uses of the distributed keyboard may also use cursor position information to allow users to edit a single, shared document.

Fig. 7. The distributed keyboard listener in action.

5 Related Work

Since the rise of CSCW, there has been a great deal of research devoted to enabling collaboration; however, current collaborative approaches are largely based on asynchronous communication such as that provided by the Microsoft Suite's Track Changes function [4]. To collaborate on a document, a group member can turn on Track Changes, then edit the document, save it, and distribute it to the rest of the group. This edit, save, send cycle leads to an asynchronous communication that effectively locks the document while someone else edits. Another common approach, simply emailing an individually edited document to another team member, also provides only asynchronous collaboration based on information exchange. Sliverware provides more synchronous communication that allows all connected group members to make and share changes concurrently.

Similar to sliverware, many new products offer synchronous editing. For example, Writely [18] provides an online collaborative word processor with the use of AJAX. There are a raft of similar solutions as part of the WebOS revolution [2] including SynchroEdit [17] and JotSpot Live [10]. While these systems are a great step forward in real-time cooperation, they have a reliance on an external server which may not always be accessible or secure. Also, software functionality can only be provided by the product team; users must trust the content provider to protect the data and add necessary features in a timely manner. Sliverware can be used to provide similar functionality for applications on

the Internet or the desktop with greater control of the application which allows for greater security and extensibility.

The aforementioned applications are written with a specific collaborative goal in mind, so while they may be useful, they are only useful for the tasks they can already perform. Sliverware is an extensible framework and has broad goals for enabling collaboration through diverse means. Sliverware shares goals with existing CSCW frameworks such as DISCIPLE and BSCW which allow users to simultaneously change documents. DISCIPLE [14] is a framework for sharing JavaBeans applications in real-time through the use of CORBA to replicate objects. BSCW provides information sharing across the world wide web with a web server that is extended using CGI scripts. DISCIPLE and BSCW provide limited usability due to the reliance on web programs; while the ideas may be extended to existing applications, the frameworks themselves cannot. Sliverware's collaborative abstractions work in desktop publishing applications as well as web applications. Sliverware focuses on a broader abstraction to enable collaboration on a larger scale.

While all the related work in collaboration is useful, the offerings do not explicitly define a simple framework for adding synchronous collaboration to existing systems across domains. Sliverware enables collaboration in a manner that is comprehensible and easily extended. Collaborative application developers have complete control of their product without spending valuable time on low level programming.

6 Conclusions and Future Work

Sliverware is designed to enable collaboration in applications through an extensible, lightweight framework that is easy to understand and easy to use. Sliverware provides the same benefits as traditional middleware, but unlike traditional middleware which provides a horizontal layer that pre-invents the wheel, sliverware focuses on thin vertical pieces of the complete application stack—network communication, group membership, and a collaborative service. Additionally, while traditional middleware provides a great deal of functionality regardless of the applications' needs, sliverware provides more focused pieces of functionality that can be optimized to ensure that the system remains as lightweight as possible.

Sliverware's extensible framework enables programmers at all levels to contribute to and use the system. Sliverware provides a framework to quickly enable collaboration in programs by furnishing a set of lightweight middleware modules; application-developers can build a system by assembling sliverware and writing minimal amounts of code to utilize the collaborative service through the API—the developer is not bogged down with low-level details and can instead focus on high-level programming.

Acknowledgments

The authors would like to thank the Center for Excellence in Distributed Global Environments for providing research facilities and the collaborative environment. This research was funded, in part, by the NSF, Grant # CNS-0620245. The views and conclusions herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

1. G. Abowd, C. Atkeson, A. Feinstein, C. Hmelo, R. Kooper, S. Long, N. Sawhney, and M. Tani. Teaching and learning as multimedia authoring: The classroom 2000 project. In *ACM Multimedia*, pages 187–198, 1996.
2. S. Adler. WebOS: Say goodbye to desktop applications. *netWorker*, 9(4):18–26, 2005.
3. Aim. `http://www.aim.com/`, 2006.
4. B. Barrios. Tutorial microsoft office word 2003: Collaboration. `http://getit.rutgers.edu/tutorials/word_collaboration/media/collaborative.pdf`, 2002.
5. K. Beck and C. Andres. *Extreme Programming Explained : Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.
6. R. Bentley, T. Horstmann, J. Trevor, and K. Sikkel. Supporting collaborative information sharing with the world wide web: The BSCW shared workspace system. *4th International World Wide Web Conference*, pages 63–74, 1995.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing Series)*. Addison-Wesley Professional, 1995.
8. J. Grudin. Groupware and social dynamics: Eight challenges for developers. *Communications of the ACM*, 37(1):92–105, 1994.
9. Q. Huang, C. Julien, and G.-C. Roman. Relying on safe distance to achieve strong partitionable group membership in ad hoc networks. *IEEE Transactions on Mobile Compututing*, 3(2):192–205, 2004.
10. Jotspot live. `http://www.jotlive.com/`, 2006.
11. J. Kuri and S. K. Kasera. Reliable multicast in multi-access wireless LANs. *Wireless Networks*, 7(4):359–369, July 2001.
12. J. Lin and S. Paul. RMTP: A reliable multicast transport protocol. In *INFOCOM*, pages 1414–1424, 1996.
13. T. W. Malone and K. Crowston. The interdisciplinary study of coordination. *ACM Compututing Survey*, 26(1):87–119, 1994.
14. I. Marsic. DISCIPLE: A framework for multimodal collaboration in heterogeneous environments. *ACM Computing Survey*, 31(2es):4, 1999.
15. C. E. Perkins and E. M. Royer. Ad-hoc on-demand distance vector routing. In *WMCSA '99: Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, page 90, Washington, DC, USA, 1999. IEEE Computer Society.
16. E. M. Royer and C. E. Perkins. Multicast operation of the ad-hoc on-demand distance vector routing protocol. In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 207–218, New York, NY, USA, 1999. ACM Press.
17. Synchroedit. `http://www.synchroedit.com/`, 2006.
18. Writely. `http://www.writely.com/`, 2006.