An architecture for building collaborative tools in Java

YC Nuckchady and J Nummenmaa

Department of Computer Science
University of Tampere
Finland
{vik, jyrki}@cs.uta.fi

Abstract. To date, there are surprisingly few collaborative applications that exploit the accessibility of the Internet. The main reason behind this is that it is rather difficult and time consuming to build such one-off applications. In this paper, we introduce an architecture built in Java that would allow rapid and easy development of collaborative applications. This architecture also provides a feature that can dynamically rewire the collaborators in order to achieve greater efficiency. This architecture was tested with Fujaba[8] case tool by enhancing some of it's single user capabilities to a level of real-time multi-user collaboration. Indeed, this approach proved to be a reliable solution to quickly building and deploying collaborative applications, and future work would be on making it more flexible and more efficient.

1 Introduction

Collaboration has existed since people have been able to communicate with each other. The advent of the internet has provided people with a communication tool that makes the great geographical distance separating people on the planet insignificant if not irrelevant. Nowadays someone can communicate an idea or a decision almost instantly via an email or an instant messenger service. Collaboration has never been so close and powerful as it is today. The world wide web provides us today with a collaboration environment platform that can span the whole globe.

To be able to collaborate on the internet one must have tools that allow him or her to have a presence on the web. To date there is a wide variety of devices that enable one to connect to the web, desktop computers, laptops, palmtops and phones. Each of these devices has its limitations in terms of memory capacity, processing speed and network connectivity speed. These factors are crucial in determining once presence on the web.

There are already some forms of collaborations on the internet, primarily file sharing and game playing. For other forms of collaboration, the participants make use of assistive technology such as news forum, chat, email and instant messaging to communicate ideas, for discussions and make decisions about a

common project. What is lacking is an appropriate software architecture that will facilitate the creation and deployment of collaboration tools.

In this paper we aim to introduce an approach to designing and building collaborative tools where data is shared and updated in real time.

2 Collaboration Model

A collaboration environment should allow any number of participants from any geographical locations that can access the Internet communicate ideas and decisions in real-time. Also, the environment should be non-discriminative with regards to connection devices and should allow for the manipulation of the collectively shared data. For the exercise of this thesis, we shall take as an example a certain number of collaborators spread over Finland, Estonia and Hungary all manipulating a mindmap (*the shared data*) using some graphical editor (*the collaborative tool*). Our collaborative environment is made up of Participants[1] and Coordinators[2]. Participants exchange information and Coordinators communicate to the Participants through the use of Capsules[3].

The type of interconnection between the participating parties (*Participants*) is illustrated in Fig. 1 below. As it can be noted, data is replicated at all participating sites and each of these sites is connected to two Coordinators, one of which serve as a backup incase the primary one fails. This type of arrangement is an extension of a centralised network configuration. The central point or point of convergence (*PoC*) is the primary Coordinator which regulates the communication among all the peers. This centralised feature is favoured over a mesh topology, as in a peer-to-peer network, because in the latter case it imposes that the communication devices should be powerful enough to sustain a high number of connections simultaneously for the duration of the session. This restriction is discriminatory and hence against our vision of a collaborative environment.

Furthermore, the interconnection topology is not rigid for the duration of a session. In order to achieve greater efficiency in terms of perceived responsiveness, the Coordinator can trigger the creation of new Coordinators and instruct selected Participants to rewire to these new Coordinators. From Fig. 2 and Fig. 3, we can observe that a rewiring of Participants *Fin 1*, *Fin 2* and *Fin 3* can achieve a better overall response time. A possible reconfiguration of the interconnection network is illustrated in Fig. 4.

3 Description of the components

3.1 Coordinator

The Coordinator operates as a seperate process in the collaborative application so that if the Participant owning the machine exits the collaborative session, the

[1] Subsection *Participant* in section *Description of components.*
[2] Subsection *Coordinators* in section *Description of components.*
[3] Subsection *Capsules* in section *Description of components.*

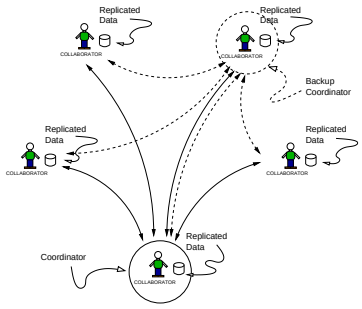

Fig. 1. Favoured topology of interconnection between participants.

Coordinator can continue without interruption. The Coordinator also behaves as an invisible participant in the session. That is, it owns a copy of the shared data and all the business rules required to update it. The Coordinator's central role is used to sychronize operations among the Participants whether through the use of timestamps, sequence numbers, or as a clock synchronizing server. The Coordinator is made up of a number of services that accomplish the specific tasks as described below.

The Main Service The Main Service listens and accepts requests to connect from new Participants. It authenticates these connections by validating a username/password pair that the participant transmits for first time. It also verifies the signature of the application for the current session. Since the Coordinator may be the same for different collaborative applications, it is necessary to inform the Participant that it might have connected to an inappropriate session. If the authentication is successful, the Main Service, hence the Coordinator sends the IP address and port number of the backup Coordinator in service. Indeed, if the session has just one Participant, then the new Participant is automatically made the backup Coordinator. In fact, the primary Coordinator triggers the creation of the backup on the machine of the new Participant if it is viable. The new Participant is also given the port number it must connect to in order to receive data update messages. Moreover, the Participant also transmits in the same bundle of data containing the username, a request for the View Update Service. This

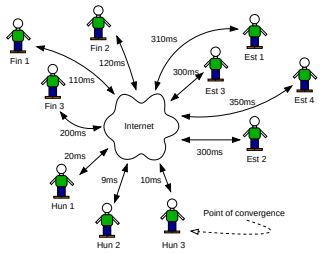

Fig. 2. A configuration illustrating the remoteness of the participants with respect to the Coordinator.

optional service relates to the order of graphical actions made by other participants. This service provides a presence of the collaborators and gives a feedback about their doings and regions of interests on the shared data. In return, the Coordinator adds to the bundle of reply data, the port of that requested service as well as the compulsory Administration and Data services. Finally, the connection is transformed to a Connection Object and is added to a queue of already connected Participants. A Connection Object is one which can be used to write to, and read from a specific Participant.

Also, when a new Participant joins the session, it must be updated with an accurate copy of the data. To this end, all distribution of update capsules is frozen and incoming ones buffered. In the mean time, the new Participant is given a serialized copy of the shared data. When the transfer has completed successfully, all the operations resume as normally. This ensures that all Participants are updated at the same time. Fig. 5 below resumes the operations of this service.

The Data Service The Data Service holds a reference to the queue of connection objects mentioned above and a reference to a queue of received data capsules. The queue of Connection Objects is periodically swept from one end to the other reading capsules that have been sent by the Participants at the other ends. The Capsules are then inserted in the queue of capsules according to the sychronization data embedded inside them. The other role of this service is to pop the ordered queue and execute the contents of the capsules. It must be

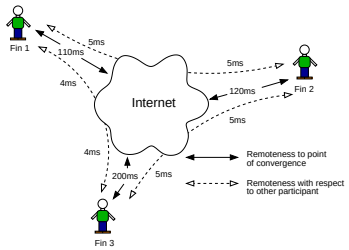

Fig. 3. Remoteness of Participants with respect to each other and the Coordinator.

stressed that this service only receives information concerning the shared data. If the embedded operation requires an update of the data, then it is executed by the Protocol Analyser Module attached to this service. In case that the operation is successful, the local copy of the shared data is updated as a consequence and the operation is broadcasted to all Participants so that the new state is reflected on their individual copies. However, if a new Participant has been accepted while the operation is executing locally, then the operation is allowed to complete on the primary Coordinator before the new Participant is uploaded a copy of the data. Subsequently, all the Connection Objects except the last one in the queue are updated with the last operation popped from the buffered queue of capsules. At this stage, all Participants including the new one would have an identical copy of the shared data. In case the operation fails for some reason, then the exception is caught by this service and only the Participant which requested that operation is informed of it's failure with some appropriate feedback. This prevents other participants from being littered with exceptions that is of no concern to them. Indeed, this approach to collaboration allows only successful update operations to be broadcasted. Fig. 6 illustrates the components and operations of the Data Service.

The Administration (*Admin*) Service This service is identical in structure to the Data Service mentioned above. It periodically polls all the Participants by requesting from them to measure the relative perceived responsiveness with

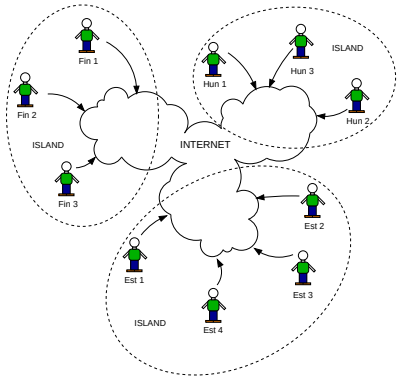

Fig. 4. Islandisation of the interconnection network into subnets.

respect to all the other Participants. This reading is a combination of the free memory footprint available in the Java Virtual Machine and the round trip time for a trivial message sent to another participant. Next, they each return to the admin service the measurements embedded in a capsule which is read by the Capsule Reader Thread shown in Fig. 6 above. The Capsules are queued. Then each of them is popped from queue and fed to an algorithm that tries to find the optimal configuration of interconnections of the Participants. If a significantly better arrangement is discovered, then the Admin Service instructs all the Participants to reconnect according to the new map of interconnections. It must be noted that there is no benefit if the islandisation process illustrated in Fig. 4 above occur too frequently. The other functionality of the Admin Service is to monitor the liveliness of all the other services, except for the Main Service which is the parent of all services. If one of the monitored services dies for some reason, this service restarts it. The services that requires monitoring can be parametrized before launching the Coordinator.

The View Update Service This service is reponsible for distributing graphical related information to those participants that have subscribed to them when they entered the session for the first time. The Main Service gives to this service a reference to the Connection Object representing that Participant. The View Update Service then adds that reference to a second list of Connection Objects, which is a subset of the ones mentioned in the Admin or Data Service. This

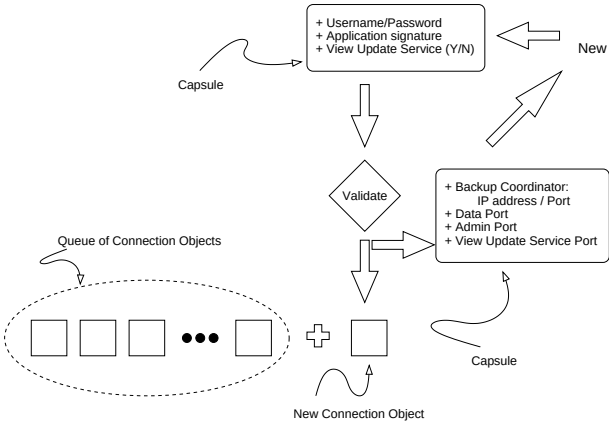

Fig. 5. Illustrating the situation when the Main Service accepts and registers a new Participant.

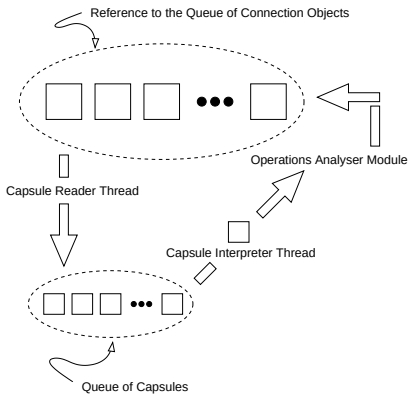

Fig. 6. Components and Operations of the Data Service.

service is a separate stream of data to those that affect the state of the shared object as not all collaborators might want to be informed of the presence of the others due to it's higher volume of traffic data. Moreover, there is no Protocol Analyser Module. In other terms, all view update capsules read are queued by a reader thread and it's counterpart, the broadcaster thread, pops the queue and publish the information to every Participant in the shorter list of references.

3.2 Participant

The Participant is the generic term for a collaborative application that can be human driven or/and automated. A Participant can be required to play the additional role of a Coordinator (primary or backup) simultaneously. Obviously that second role is a totally independent role to that of the Participant. The scope of a Participant is defined by the core plugin which is intricate to the Collaborative Architecture and collaborative plugin. It's interaction is as transparent as interacting during a single user mode session.

The architecture of the Participant is built along the Model-View-Controller (MVC) software design pattern. Fig. 7 illustrates the components making up a Participant. As it can be noted, the controller is made up of two parts, the application module and the communication module. The Controller always communicates to the Coordinator the messages caught from the View. If the Coordinator approves of it, then the Controller informs the Model of the change in state and it in turn informs the View which takes appropriate visual update actions.

Each Participant is connected to the Coordinators through two main *lines of communication*. One is used for *data* exchange concerning the shared object and the other line is used for *administrative* purposes. It must be noted that administrative operations are transparent to the Participants as there is no interaction with the collaborative application itself. The Data line can be made up of one or two streams depending on whether the Participant is connected to the View Update Service.

The Participant starts by requesting a connection to a well-known internet address and port where the primary Coordinator of a session is hosted and listening on respectively. As soon as the socket connection is established, the participant transmits the following pieces of information:

- A Username/Password pair that will be used to authenticate the Participant and serve as an ID to inform other Participants of it's presence.
- A signature of the collaborative application, so that a match can be made by the primary Coordinator overseeing the session that this Participant wishes to join. The motivation is that the Coordinator is part and parcel of the Collaborative Architecture and hence is the same for all applications. The distinction between the collaborative applications is made by the difference in the library of plugins that they use. Hence it is deemed important to transmit the ID of the application to the Coordinator so that the Participant can be informed already at the start of a mismatch in the session it wants to connect to.

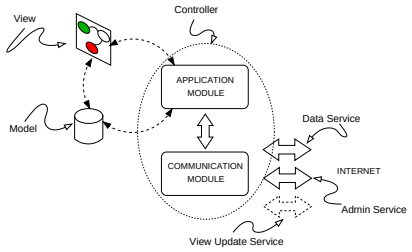

Fig. 7. Software design of the Participant.

– A request to connect to the View Update Service mentioned above. At this stage, a Participant is bombarded with all graphical updates allowed by the applications and generated by all participants. The only allowed graphical action that is distributed is when a graphical token is dragged on the screen. It was deemed sufficient to demonstrate the presence of a Participant and it's current region of interest. The request is made in terms of a remote method invocation approach. This allows flexibility.

As mentioned above, the Coordinator responds with appropriate data if the authentication has proved successful. The capsule sent by the Main Service in the Coordinator is buffered in a queue in case too much comes into the participant while it is still processing one capsule. Anyhow, the Capsule Interpreter Thread in the Participant pops the queue of capsules and sends it for processing to the attached Protocol Analyser Module. There the appropriate objects in the Application Module are called and the View or the Model is passed on with the relevant information. Fig. 8 illustrates the components of the Communication Module of the Controller and the flow of information through it.

The Application Module is made up of two sets of plugins. The first set is the default used for administrative purposes. For example, calculating the amount of free memory space available to collaborative application or for connecting to backup Coordinator. The other set concerns the application itself. The main concern when designing the plugin is to make those operations that alter the state of the shared data transactional. This can be achieved by defining a transaction

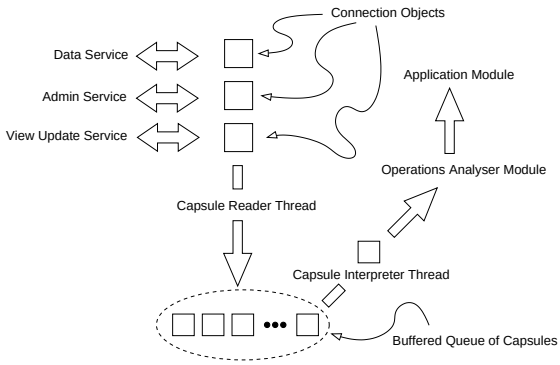

Fig. 8. Components and operations performed by the Communication Module.

as a series of commands that generates exceptions. The Protocol Analyser is designed such that if a Command Object is executed and one of the method calls throws an exception, then the execution of the Command Object is interrupted and the Protocol Analyser informs only the Participant who initiated the transaction of the failure. However if the transaction succeeds at the Coordinator's site, then the Protocol Analyser distribute this transaction to all Participants.

3.3 Capsule, Command Object and the Protocol Analyser Module

A Capsule contains a Command Object and propagation field which indicates if that Command Object is to be distributed to everyone or intended to the receiver of that Capsule. The Command Object is a set of method calls to one or many objects. It is represented by delimited strings and it's structure is described below,

Table 1. The syntactic structure of a Command Object.

<Data Object>:=	<Command String>
<Command String>:=	<Fragment>|<Fragment>C_S
<Fragment>:=	<Command Name>M_S<Method Name>|
	<Command Name>M_S<Method Name>A_S<Argument>
<Command Name>:=	A valid class name according to Java's Language Specification
<Method Name>:=	A valid method name according to Java's Language Specification
<Argument>:=	Simple Data Types according to Java's Language Specification |
	<Argument>A_S
C_S :=	Command Separator
M_S :=	Method Separator
A_S :=	Argument Separator

The Protocol Analyser Module takes as input an Command Object and decomposes it into Fragment of commands. It then feeds each of them to an execution engine made up of objects from the `java.lang.reflect` package. All arguments to a method call are passed as `java.lang.String` data types. The invoked method has to make the appropriate data type conversion so that the argument can be used as intended. Thus, it is the responsibility of the plugin designer ensure that these constraints are implemented. Also as indicated above, methods that form part of a transaction should be described to throw an `fi.uta.ctk.protocol.CommandException`. This exception is used to break the loop of fragment execution in the Protocol Analyser. Furthermore, the latter maintains a dynamic list of Objects that have already been created. At this stage the objects that are invoked typically contains accessor methods to the shared data. When a request to one of these methods is made, the execution engines checks in it's list for an existing reference. If there is one, it is used, otherwise a new instance is created and a reference to it is added to the list of active objects.

This strategy eliminates the repetitive creation of the same objects each time an invokation is made.

4 A case study and Results

This Collaborative Architecture was used to implement a collaborative version of Fujaba in the context of an editor for UML class diagrams. Fujaba is by nature a single user application and hence already has the infrastructure for the storage of data. This fulfills the requirement of having data replicated at all participating sites according to Fig. 1.

The first step in enhancing the Fujaba application to the level of a collaborative tool was to identify those operations that affected the state of the shared UML diagram and those that could be used to indicate the presence of a Participant to it's peers. In this case it was decided to have two such operations, one whereby if an object is locked for update by a participant, then that object appears of a different colour than the others. The other graphical operation is to display to all Participants and at all times the movement of classes when they are dragged on the screen. For both these operations, the ID of the responsible Participant is made visible when the mouse is posititioned over these graphical objects.

Table 2 gives an example of an operation that is part of a transaction. As it can be noted, the method `setBalance(float)` throws an exception as it alters the state of a piece of data. Also note that the method `getBalance(String)`, takes a string parameter. This argument is sent in the Fragment that is currently being processed so that `getBalance` knows to which object it must send back this result. The `callingMethod` is of the form of a fragment and `balance` is appended to it.

The enhanced version of the Fujaba case tool behaved smoothly in a collaborative session of 5 participants at a network distance of 3ms from the Coordinator. However to determine the stress level of this architecture, a barebone version of the environment was used with an automated chatting program. The choice of this type of application made it very easy to tune the rate of flow of information between Participants and Coordinator. The experiment aimed to determine a balance between the number of users and the rate of flow of messages between the parties. Indeed, in a session like the one above, there are two types of updates, distributed graphical updates and distributed data updates. Typically, the former generate more messages than the latter. The experiment was made up of two sets of four sessions. In each set, the rate of flow of messages was kept constant and the number participants varied for the each of the sessions. The number of participants were carefully chosen so that a wide spectrum of readings could be obtained. The first set of measurements was tested under a rate flow of 20 messages per second and the second one was tested at a rate of 2 messages per second. These rates are the estimated upper limits of number of generated messages for the two types of activity identified above. In other terms this means that for example in the case of data update, a human user

Table 2. An example of a command object written for a plugin.

```
 1 package fi.uta.ctk.protocol.command;
 2 import fi.uta.ctk.protocol.CommandObject;
 3 import fi.uta.ctk.protocol.ProtocolException;
 4
 5 public class Balance
 6 {
 7    private float Balance;
 8
 9    public String getBalance(String callingMethod)
10    {
11       return callingMethod+CommandObject.ARGUMENT_SEPERATOR
                +Balance;
12    }
13
14    public void setBalance(String amount) throws
                ProtocolException
15    {
16       try
17       {
18          float value = Float.parseFloat(amount);
19          if(value < 0)
20             throw new ProtocolException("Balance cannot be less
                        than 0");
21          this.balance = value;
22       }
23       catch(NumberFormatException nfe)
24       {
25          throw new ProtocolException("Invalid Balance Format");
26       }
27    }
28 }
```

can at maximum generate 2 transactional upates per second. The experiment was repeated over a number of days and at different times of the day. In this experiment, we monitor the growth of the size of the queue of capsules (buffer) at the Data Service over a period of four minutes.

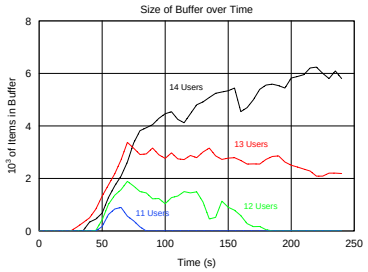

Fig. 9. The rate of flow of messages is 20 per second and is maintained over 240 seconds. Each plot indicates the growth of the queue size for that number of Participants involved.

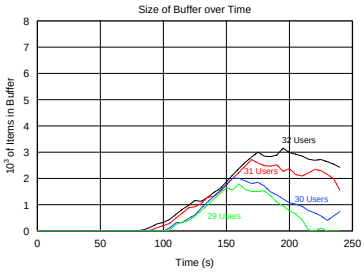

Fig. 10. The rate of flow of messages is 2 per second and is maintained over 240 seconds. Each plot indicates the growth of the queue size for that number of Participants involved.

Fig. 9 clearly demonstrates that for each session, past a critical value the time required for the buffer to empty grows exponentially with time. This effect is also present in Fig. 10 but the time required is not as large as in the case of the view update data. This is because the queue size is not as large. In both cases, these types of sessions is not recommended as it's quality would degrade very quickly to the extent that it may take about 30 seconds for the view update service and 80 in the case of data update. Moreover, under normal operational circumstances it has been estimated that a session of 30 participants can be easily handled.

5 Conclusion and Future Work

In this paper, we proposed an architecture for developing collaborative applications. A centralized topology of interconnection is favoured and it is improved by

a backup PoC. This type of arrangement makes it easier to synchronize Participants with respect to each other. We also saw that a Coordinator serves as the PoC and it is made up of a number of four services, the Main Service, the Data Service, the Admin Service and the Visual Update Service. The Participant is designed according to the MVC software pattern. The Controller part is made up of two parts, the Communication Module and the Application Module. The latter consists of the application itself and two sets of plugins, the core plugin intricate to the architecture and the application plugins. The objects in the latter set are essentially objects that update the shared data and what can be used to indicate the presence of a collaborator to it's peers. The data update objects are first executed at the Coordinator's site and only if it is successful that it is broadcasted to all the Participants. Otherwise the exception is caught and signalled only to the responsible Participant.

One feature that can be considered for the future is a strategy to select the regions of interests for the View Update Service and also the type of graphical updates that one is willing to subscribe to. This will then prevent any collaborator from being bombarded with irrelevant graphical update information.

6 Acknowledgements

This work has been carried out in the UML++ project funded by the Academy of Finland.

References

1. Jari A. Lehto, Pentti Marttiin, Nokia Research Center: Lessons Learnt in the Use of a Collaborative Design Environment. 33rd Hawaii International Conference on System Sciences-Volume 8
2. Doug Lea: Concurrent Programming in Jaba(TM): Design Principles and Pattern(2nd Edition)
3. Jim Farley: Java Distributed Computing
4. Andrew S. Tanenbaum and Maarten van Steen: Distributed Systems: Principles and Paradigms
5. Randy Chow and Theodore Johnson: Distributed Operating Systems and Algorithms
6. Sun Microsystems Inc.: Java Shared Data Toolkit. http://java.sun.com/products/java-media/jsdt/
7. Sun Microsystems Inc.: Java Remote Method Invocation. http://java.sun.com/docs/books/tutorial/rmi/
8. University of Paderborn, Software Engineering Group: The Fujaba Project. http://www.uni-paderborn.de/cs/fujaba/