A Toolkit to Support Distributed Fault-Tolerance Algorithms

Letao Jiang
Department of Computer Science
The University of North Carolina at Chapel Hill
Email: jiang@cs.unc.edu

Abstract. A distributed program is presented, which functions as an instant message application, and at the same time, ensures the delivery order of messages to the functioning processors in a P2P network, in spite of the fact that processes may crash at any time. Fault-tolerant broadcast algorithms in distributed systems have been proposed for years but rarely have an intuitive way to view it. The instant message application provides a graphic user interface that gives the ability to display how the algorithm works. It also visually demonstrates the fault-tolerant broadcast algorithm can be implemented in a modular fashion using an accompanying toolkit. An efficient reliable broadcast protocol is used. It is assumed that all processor failures will be detected and result in halting the failure processor.

1. Introduction

The invention of distributed systems has changed the entire world and people's daily life. When people use the web browser and attempt to connect to a web server, they are participating in what seems to be a simple form of a client/server distributed system. When people use a web application provided by modern servers such as Google or Facebook, they do not just interact with one single machine, these applications are built from a large collection of machines. It is possible that machines may fail at any time, and people do not want to see one machine failure bring the entire application down. Fault-tolerance is one of the central challenges in constructing distributed systems. Many kinds of research have been done on fault-tolerant algorithm in distributed systems, but people except the researchers may not have a direct view of the algorithm. A distributed system is needed in order to illustrate the fault-tolerant broadcast algorithm.

The precondition of a fault-tolerant distributed system is atomic broadcast. A well-designed atomic broadcast algorithm will carry the benefit of fault tolerance and increase the likelihood of building a fault-tolerant distributed system. The first goal we want to achieve is to implement atomic broadcast in our application.

An atomic broadcast algorithm can have different variants and each variant shares some commonalities. A shared library can streamline the implementation of each variant by providing the common API for the same algorithm. Our second goal is to create such a shared library that can exploit the commonality among the variants of the targeted atomic broadcast algorithm.

Our third goal is to separate algorithm and code by giving the user a direct view of the algorithm while the application is running.

The rest of the paper is structured as follows: Section 2 discusses the background on which the application is based. Section 3 presents the design or API of the application. Section 4 shows demos of the application given different scenarios. Section 5 talks about conclusions and

direction for future enhancement.

2. Background

2.1 Fault tolerance

The phrase "*fault-tolerance*" is a combination of two distinct words, "fault" and "tolerance". "Fault", in the system point of view, can be described as a malfunction or deviation from expected behavior. The meaning of "tolerance" is the ability to enable or accept something that is harmful. Fault-tolerance refers to a system's ability to deal with or endure malfunctions [1]. It is the property that enables a system to continue operating properly in the event of the failure of some of its components.

Fault-tolerance in a non-distributed system is essential because it allows the computer keeps executing with the presence of defects. The general approach to building fault-tolerant system is redundancy [9]. There are three different kinds of ways: information redundancy, time redundancy, and physical redundancy.

Information redundancy uses replicated or coded data to achieve fault tolerance. For example, Hamming code, by providing an extra bit in the data, can recover a certain ratio of bit fault [10]. Other examples that use this technique are parity memory, ECC (Error Correcting Codes) memory, and ECC codes on data block [1].

Time redundancy provides fault tolerance through performing an operation several times. Timeout and retransmissions in reliable P2P network and group communication are examples of time redundancy. Time redundancy can only be used that transient or intermittent faults occur. When permanent faults occur, time redundancy can be useless.

Physical redundancy is used in the physical level instead of the data. By adding more components to enable the system's ability to deal with faults. RAID 1 (Redundant Array of Independent Disks) is an example of physical redundancy.

When addressing physical redundancy, we can differentiate redundancy from replication. With replication, we have several units operating concurrently and a voting system to select the outcome. With physical redundancy, only one unit is functioning while the redundant units are standing by to fill in in case the unit ceases to work.

One example of well-known systems that combine the fault-tolerant techniques is called the Tandem Computer System. Systems whose fault tolerance is based solely on hardware mechanisms can be designed to provide high reliability and continue to operate in the presence of hardware component failure. However, a very large percent of computer system crashes are caused by software [2]. Tandem system provides fault-tolerant feature from both hardware and software.

Each component in the Tandem system is designed to tolerate faults. For instance, one of the key fault tolerance mechanisms that can provide the ability to tolerate both single hardware fault and most transient software faults is called *process pair*. The basic idea of process pair is to pair two processes, a primary process, and a backup process. The primary process will do all the work and the backup process, which is passive but is prepared to take over when the primary process

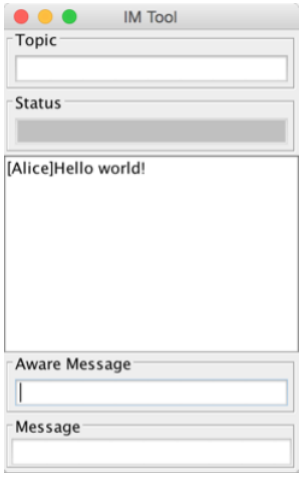


Figure 1. GUI for application

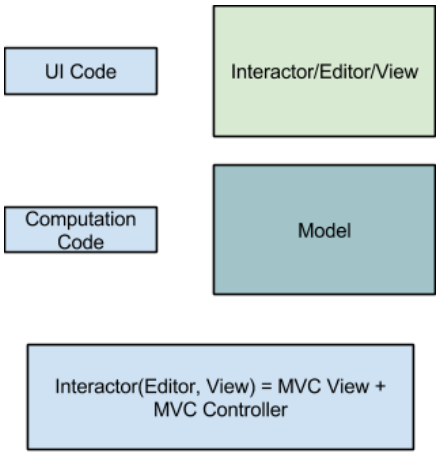


Fig 2. Model/Interactor Pattern

fails. Most faults in production software are transient [8]. When the primary process fails, programs can continue execution in the backup process if the software bug is transient.

Fault tolerance technique can be applied to different systems such as hardware system, mobile system, software system, and distributed system. Our goal is mainly focused on fault tolerance in distributed systems.

2.2 Distributed System

A distributed system is a collection of independent computers that appears to its users as a single coherent system [9]. There are some important aspects that need to be mentioned in the definition of distributed systems. The first one is that a distributed system consists of components (i.e., computers) that are autonomous and the second one is that each single user thinks they are dealing with a single system [9]. One example of distributed systems is the program I built that can function as an instant message application.

This application provides the function of instant message (IM). Each user can use the application as a single system. Users will join or create a session when they use the system. By joining the same session, two or multiple users can communicate using the graphical user interface (GUI). Figure 1 illustrates the GUI of the application. For the IM application, there are two text fields that users can use to type words. When a user is typing in the aware-message text field, other users in the same session will notice that others are typing from seeing the display text in the status. A demo of the application can be found in the appendix (App. 8).

We use model-view-controller (MVC), which is a software architectural pattern for implementing user interfaces, to implement the application. The central component of MVC, the model, captures the behavior of the application in terms of its problem domain, independent of the user interface [11]. The model directly manages the data, logic and rules of the application. A view can be defined as any output representation of information. The third part, the controller, accepts input and converts it to commands for the model or view [12]. MVC architecture can vary significantly from the traditional definition. Figure 2 illustrates the pattern I used for my application. The IM application uses a string list as the model while the editor application uses a character list. We combine the view and controller into the GUI and let the GUI take input or display output of the model. We call the combination of the view and controller an interactor.

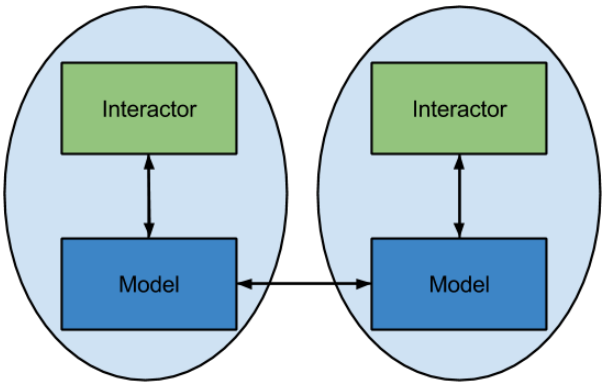


Fig. 3 Replicated Architecture

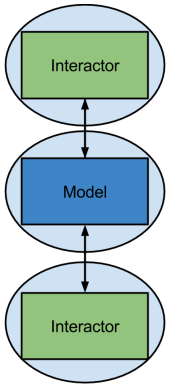


Fig. 4 Centralized Architecture

Since the application is a distributed system, we will need some mechanism to distribute the model to all users. There are two approaches to sharing a model, replicated and centralized model sharing. Figure 3 and Figure 4 illustrates the two different architectures of model sharing. Two types of model sharing architectures both have their pros and cons. Centralized architecture has the disadvantage that it may have round trip delay to get local feedback or the central server may not be available at any time while the replicated architecture has the problems of consistency issues of causality and concurrent operations and correctness and performance issues when model accesses central resources, is nondeterministic, and has side effects. We choose the replicated architecture as the solution to the model sharing because the application is an IM application, and we don't always want to wait for possbily long round trip delay. An efficient broadcast mechanism can support causality and resolve the consistency issue caused by concurrent operations.

2.3 Failure Type

One important characteristic of distributed systems that separates them from single-machine systems is the notion of a partial failure [9]. A partial failure happens when one of the components in the distributed system fails. In non-distributed systems, a failure often happens in total, bringing the system down. In contrast, failure in distributed systems could affect only one component but leave the rest of the components unaffected.

It is important to characterize the failures when one is trying to figure the problems of failures in a distributed system [5]. The general classes of failure are the following.

- *Crash failure*. A process halts but is working correctly until it halts.
- *Omission failure*. A process does not respond to incoming requests.
  - *Receive omission*. A process fails to receive incoming messages.
  - *Send omission*. A process fails to send the message.
- *Timing failure*. A process's response lies outside the specified time interval.
- *Arbitrary failure*. A process may produce arbitrary responses at arbitrary times.

More specifically, a *crash failure* will force an ongoing operation, usually send or receive, to terminate and perform no further operation. An important aspect of crash failure is that once the process is halted, nothing can be heard from it. An example in our system point of view is that a

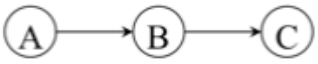


Fig 5. No redundancy

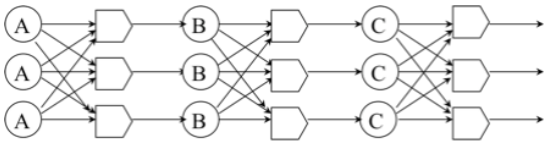


Fig 6. Triple Modular Redundancy (TMP)

process can continue receiving or sending messages until time t when a crash failure occurs in the process.

A process can undergo two different omission failures, send omission and receive omission. Processes in send omission cannot send messages to other processes, but can still receive messages and perform the action it receives. Send omission failures are commonly caused by a lack of buffering space in the operating system or network interface, which can cause a message to be discarded after the application program has sent it, but before it leaves the sender's machine [16]. A receiving omission is the converse. Receiving omission failures occur when a message is lost near the destination process, often because of a lack of memory in which to buffer it or because evidence of data corruption has been discovered. Omission failure exhibits send omission, receive omission, or both [1, 4, 9].

Timing failure is failures where a message breaks some time sensitive contract between processes [9]. The message will be lost if a message arrives too early to a process, but the receiving buffer will not be available until a certain time.

Arbitrary failures are the most serious type of failure, and also known as Byzantine failures. In our example, if 'A' process undergoes arbitrary failures and wants to send "lunch?" to the other processes, then it may send a "done?" to other processes instead [9].

In the system that we presented, we assume that only crash failure will occur.

2.4 Approaches to Fault Tolerance in Distributed Systems

The general approach to achieving fault tolerance in distributed systems is replication. There are two ways to approach such replication: *active replication* or *primary backup*.

Active replication is a technique for achieving fault tolerance through physical redundancy. A common instantiation of this is triple modular redundancy (TMR) [1]. For example, consider a system in which the output of 'A' goes to the input of 'B' and the output of 'B' goes to the input of 'C' (Figure 5). Any single point of fault will bring the whole system down. In TMR design, all the components will be replicated three ways and place voters after each stage to pick the majority outcome of the stage. The voter will select the majority of the input and produce the output. The voters themselves are replicated because they too can be faulty (Figure 6).

If TMR is used to design our system, each interactor and model will be replicated three ways. We can treat 'A' as the interactor, 'B' and 'C' are the models. If 'A' entered an input, three replicated processes will both output the operation to the voter, and the voter will select the majority of the input and produce an input to 'B'. Despite two of process 'A' halting or one of the processes 'A' producing a random output, the input goes to 'B' will always be the correct one. Triple replication of 'B' will all get the same input operation and produce an output operation to the voter. TMR can handle 2-falut tolerances with crash failure or 1-fault tolerance with arbitrary failure.

With a *Primary backup* approach, one process (the primary) does all the work. When that process fails, the backup takes over. We used this approach to tolerate the sequencer process failure in our implementation.

An implicit precondition for the replication approach is that all messages arrive at all process in the same order, also called the atomic broadcast problem.

2.5 Reliable Broadcast

The design and verification of fault-tolerant application is widely viewed as a complex endeavor [3]. The main reason that leads to the case is the weakness of the available communication primitives in the distributed application. For example, many systems can provide the primitive that allows a process to send a message to one process at a time. If a process p wishes send a message m to other processes in the network, it will need to send the message multiple times. What if p fails in the halfway of the sending process, it may cause some of the processes receive m but others do not [3]. Arising of the inconsistency complicate the development of fault-tolerant distributed system.

Fault-tolerant broadcasts communication primitives increase the likelihood of building fault-tolerant applications. The weakest type of fault-tolerant broadcast is reliable broadcast and it does not ensure any order. Reliable broadcast only ensures that all correct processes will eventually deliver a message that is broadcast by a correct process. Stronger variants of the reliable broadcast will force more requirements on the order of delivery of the messages [3].

There are three different orders of messages, *FIFO order*, *Causal order*, and *Total order*. By forcing the delivery order of the message to the reliable broadcast, three variants of the reliable broadcast can be proposed, FIFO reliable broadcast, Causal reliable broadcast, and Atomic reliable broadcast. Figure 10 [4] shows how all the broadcast variants can be derived from the basic broadcast type – reliable broadcast.

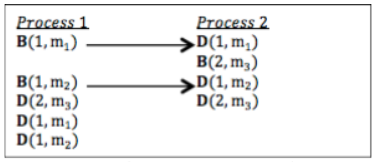


Fig 7. FIFO order but not Causal

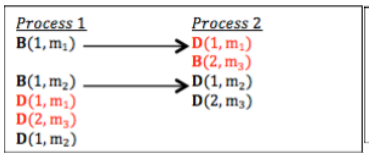


Fig 8. Causal order but not Total

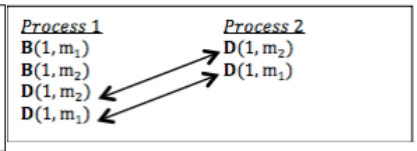


Fig 9. Total Order

B(1, x) means broadcast message x from process 1 D(1, x) means deliver message x to process 1

FIFO order. If a correct process broadcast message m_1 before message m_2, then no correct process delivers m_2 before m_1 (Figure 7 [17]). FIFO reliable broadcast guarantees the FIFO order of message delivery, and always supported in the TCP/IP protocol.

Causal order. If a correct process broadcast a message m_1 that causally precedes the broadcast of a message m_2, and then no correct process delivers m_2 before m_1 (Figure 8 [17]).

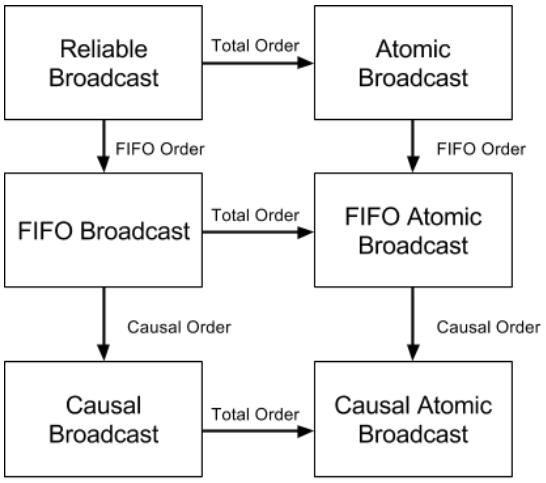


Fig. 10 Broadcast Hierarchy [9]: All broadcast variants are derived from Reliable broadcast.

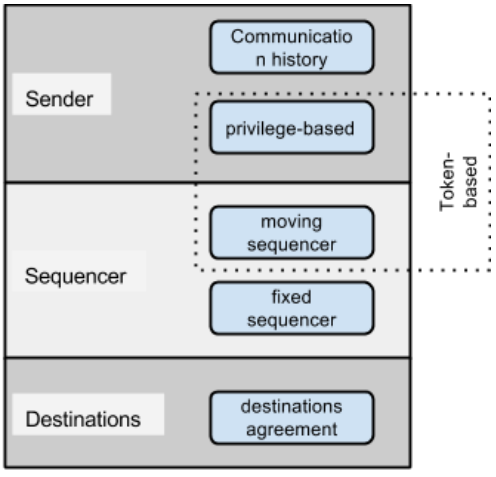
Causal reliable broadcast is another variant of reliable broadcast, which strengthens the FIFO broadcast. Sometimes FIFO order is not enough because FIFO ordering is too limited in context (a single process). For example, User 'A' sends a paper to User 'B' and 'C', User 'B' sends another paper that can be understood after reading the paper sent by 'A' first. User 'C' gets the paper sent by 'B' first and replying with his understanding of the paper sent from 'B', which is a misunderstanding of the actual meaning. The above problem could occur when the system is using a FIFO reliable broadcast. By introducing causal order to the system, it could solve the above problem. The paper sent by User 'B' will not be delivered to User 'C' until the paper sent from User 'A' is delivered to User 'C'.

Total order. If correct processes p and q both deliver message m_1 and m_2, then p delivers m_1 before m_2 if and only if q delivers m_1 before m_2 (Figure 9 [17]).

Causal order imposes a partial ordering in the system, so messages without causal relationships are logically concurrent and do not have any delivery order guarantees. This can be problematic in some cases. Consider the case of two replicate articles. Process a broadcast message m_{1a} "remove paragraph 1". Process b broadcast message m_{1b} "remove paragraph 2". If the messages deliver to the first article in the order of m_{1a}, m_{1b} and deliver to the second article in the order of m_{1b}, m_{1a}, then the first and third paragraph of article 1 will be removed, but the first and second paragraph will be removed in second article. Here we wish that all messages delivered to all correct process in the same order. Total order broadcast, also known as atomic broadcast, is another type of reliable broadcast that can ensure the above property.

2.6 Atomic Broadcast Algorithm

There are different algorithms for atomic reliable broadcast. Before we introduce the algorithms, we will first define the notation will be used in the algorithm. A sender process is process p_s from which a message is sent. A destination process is a process p_d to which a message is sent. Finally, a sequencer process is not necessarily a sender or a destination but is somehow involved in the ordering of messages [4]. It is possible that a process takes several roles simultaneously, however, the meaning of each role is totally different from each other.



Classes of atomic reliable broadcast algorithms

Depending on the three different roles that a

process can take, atomic broadcast can be categorized into three basic classes. This allows us to define subclasses as illustrated in the above figure. This results in the five classes illustrated on the figure: *communication history*, *privilege-based*, *moving sequencer*, *fixed sequencer*, and *destination agreement*. Our implementation used the fixed sequencer algorithm as the atomic broadcast algorithm.

2.7 Fixed Sequencer Algorithm

In a *fixed sequencer algorithm*, one process is elected as the sequencer and is in charge of ordering messages. The sequencer is unique and fixed, and will not normally transfer the responsibility to another process unless it crashes. Figure 11 illustrates the mechanism of a fixed sequencer algorithm. The solid black circle in the figure represents the sequencer.

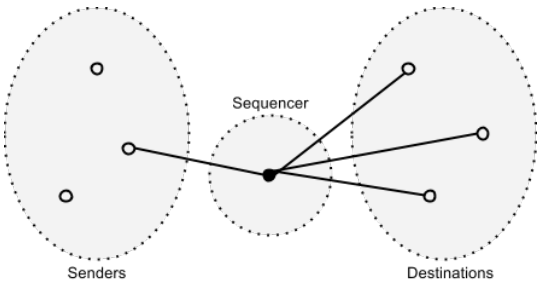


Fig. 11 Fixed Sequencer Algorithm

The pseudo-code (see appendix page 17, 18 and 19) illustrates the approaches to the algorithm. One process takes over the role of sequencer and builds the total order [13]. Those approaches did not have the fault tolerance mechanism in it so they cannot tolerate failure of the sequencer.

Fixed sequencer algorithm can have three different variants. Figure 12 illustrates some common variants of fixed sequencer algorithm ('U' stands for unicast and 'B' means broadcast.). Then those variants of fixed sequencer algorithms can be easily understood. The first variant "UB"("Unicast-Broadcast") consists of a unicast to the sequencer, followed by a broadcast by the sequencer, which is the simply fixed sequencer algorithm we introduced above [5]. The second variant "BB" (Broadcast-Broadcast) consists of a broadcast to all the destinations from the sender followed by a second broadcast from the sequencer [6]. The last variant "UUB" (Unicast-Unicast-Broadcast) consists of a unicast from to the sequencer and the sequencer replies with a unicast to the sender followed by a broadcast to all destinations from the sender.

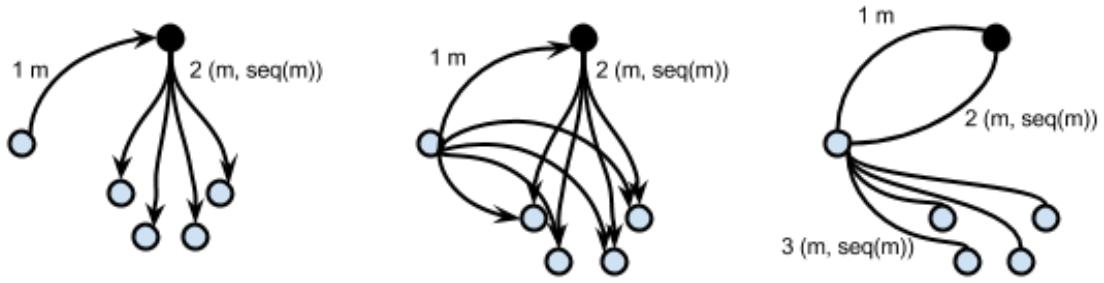


Fig. 12 Common variants of fixed sequencer algorithms

More specifically, in UB variant, if process 'A' wants to broadcast a message *m*, 'A' will first send *m* to the sequencer. Upon receiving *m*, the sequencer assigns it a sequencer number and wraps it into *m* to the destinations. Destinations then deliver messages in sequence thus according to the total order built by the sequencer. Pros: The UB variant requires only a unicast and a

broadcast, so the total messages needed to broadcast a message is n+1 (n is the number of clients), which is the least among the three variants. Cons: The sequencer will do all the broadcast jobs so there exists threshold of the sequencer's sending speed.

In BB variant, if process 'A' wants to broadcast a message *m*, 'A' will first broadcast *m* to all destinations. Upon receiving *m*, the sequencer assigns it a sequence number and computes the hash code of *m* while the destinations will put *m* into a map using the hash code of *m* as key. After that, the sequencer wraps the sequence number and the hash code of *m* into another message *m'* then broadcasts to all destinations. Destinations then deliver message according to the sequence number. Pros: Sequencer only need to send the hash code of the message instead of the message body, it improve the speed of sending. Cons: It requires the most number of messages to broadcast which is 2*n since BB variant need two broadcasts.

In UUB variant, if process 'A' wants to broadcast a message m, 'A' will first send m to the sequencer. Upon receiving m, the sequencer replies the sender with a sequence number. The sender then wraps the sequence number into m and broadcast m. Destinations deliver message in sequence based on the sequence number. Pros: The UUB variant will not be affected by the sending speed of sequencer. Cons: If latency between a user and the sequencer is too high, and when the user tries to send a message, it will need wait for the round trip delay to broadcast the message.

One of the cores of fixed sequencer algorithm is the sequence number. The total order of messages is built according to the sequence number. The difference maybe hard to see without the sequence number in UB variant, since the sequencer is responsible for broadcast the messages and the messages will always deliver in the same order. However, in other two variants, clients need to broadcast the messages, it is possible that all the messages deliver to destinations in the different order due to the latency among clients.

2.8 Multicast Library

As I mentioned before, one of our goals is to exploit the commonality between those three variant of fixed sequencer algorithm using a shared library. A shared library typically means the same pieces of code that can be reused by different programs or application. This application is built on top of the multicast library created by Professor Dewan at UNC-Chapel Hill. I will present some functions in the library that I used in my application. Join method, this method is used when a new client joins the current session. When this method is called, all the session message listeners will be notified and the clientJoined method in the listeners will be triggered. Leave method, this method is used when a client leaves or a process crashes. When the leave method is called, all the session message listeners will be notified and the clientLeft method in the listeners will be triggered. ToOthers, toClient and toAll method are used to communicate between processes. As the name described, toOthers method will send a message to other processes except itself, toClient method will send a message to a specific client, and toAll method will broadcast a message to all processes. Those functions provided in the library simplify the step of building this distributed system. A video link in the appendix illustrates a concrete example that using the multicast library to achieve causality [14] (App. 4).

2.9 Traceable Algorithm

An algorithm is needed for understanding a program, testing and debugging but in most of cases, the algorithm is tightly tied to the implementation. So one of our goals is to separate the algorithm from the application. Using in-line algorithm can extract comments from the code to view algorithm, but it does not get a linear path from scattered object. We can get the linear path by adding print statements in the program, however, one cannot disable them easily and it will contaminate the output since it will be hard to separate the statement from the output. A debugger can be a good way to trace the algorithm but it also has some disadvantages such as testing a race condition is very difficult using a debugger, the breakpoints cannot be transferred to another computer. A traceable algorithm provides an elegant way to separate the algorithm from the program. It provides an executable specification of algorithms implemented with code. It allows algorithm steps to be in separate packages, filtering by event type and provides ways to find implementations of a step [14]. A video that illustrates more details of traceable algorithms can be found in the appendix (App. 5).

3. Contribution

To achieve our first goal, which is implementing atomic broadcast in our application, we firstly applied the fixed sequencer algorithm into our distributed system so that the IM application can broadcast messages in total order. Figure 13, 14 and 15 illustrate the semantics of fixed sequencer algorithm in the context of our application.

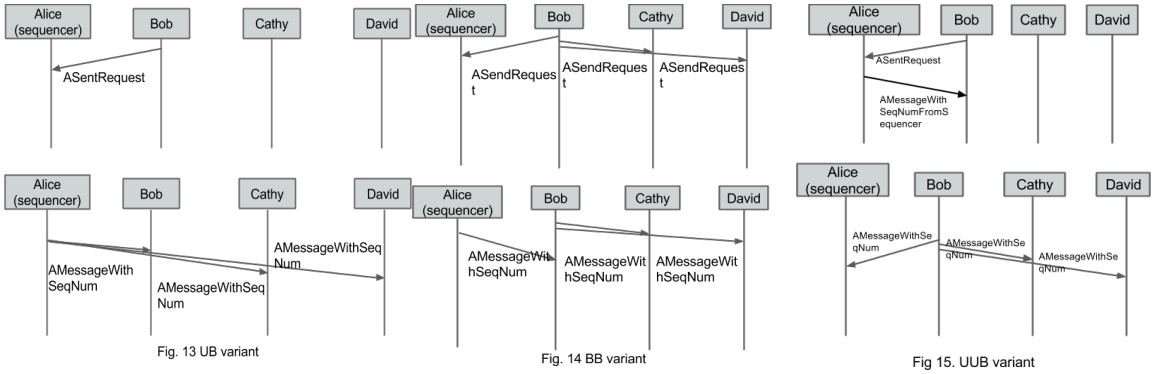


Fig. 13 UB variant

Fig. 14 BB variant

Fig 15. UUB variant

One of the most essential components in the application is called “FTManager” that is in charge of the election of the sequencer, managing the order of the message, recovering the history when new client joins, and the re-election of the sequencer. Figure 17 presents the API for the “FTManager”.

As illustrated in figure 17, a FTManager is a SessionMessageListener and a PeerMessageListener. When a user joins a session, the two listeners need to be registered in the communicator by calling the multicast library function addSessionMessageListener and addPeerMessageListener. After registering in the communicator, functions in SessionMessageListener will be invoked when join or leave function is called by the library and functions in PeerMessageListener will be invoked when any of the three functions, toAll, toOthers, and toClient, is called by the library.

```
public interface FTManager extends SessionMessageListener, PeerMessageListener {
  SentRequest warpSentRequest(ListEdit<String> listEdit);

  ListEdit<String> unwrapSentRequest(SentRequest aSentRequest);

  MessageWithObj wrapMessage(ListEdit<String> listEdit, String variant);

  void recover(String aClientName);

  void electSequencer(String aClientName, boolean isNewSession, Collection<String> allUsers);

  void reElect();
}
```

Fig. 17 API for FTManager

We need to introduce a new component when talking about the initialization of FTManager, ARegistry. ARegistry class is a linking between communicator and FTManager since there could exist multiple instances of communicator, each communicator needs a FTManager. We use ARegistry to find out whether the corresponding FTManager exist or not by giving a communicator (Fig. 18). If the FTManager exists when looking up using an instance of communicator, it will use that FTManager to send the message, otherwise, ARegistry will create a FTManager and link the communicator with it, then send through it.

```
public static FTManager getFTManager(Communicator communicator) {
  if (hct.get(communicator) == null) {
    FTManager ftManager = new AFTManager(communicator);
    hct.put(communicator, ftManager);
    communicator.addSessionMessageListener(ftManager);
    communicator.addPeerMessageListener(ftManager);
  }
  return hct.get(communicator);
}
```

Fig. 18 Method to link FTManager with communicator in Registry

After the initialization, users can use the IM application to broadcast messages either with or without total order. By checking the checkbox before “total order” (Fig. 19), the FTManager can switch from non-total order to total order. There is a Boolean variable in FTManager that control whether total order will be used and a function, isFT, can return that variable.

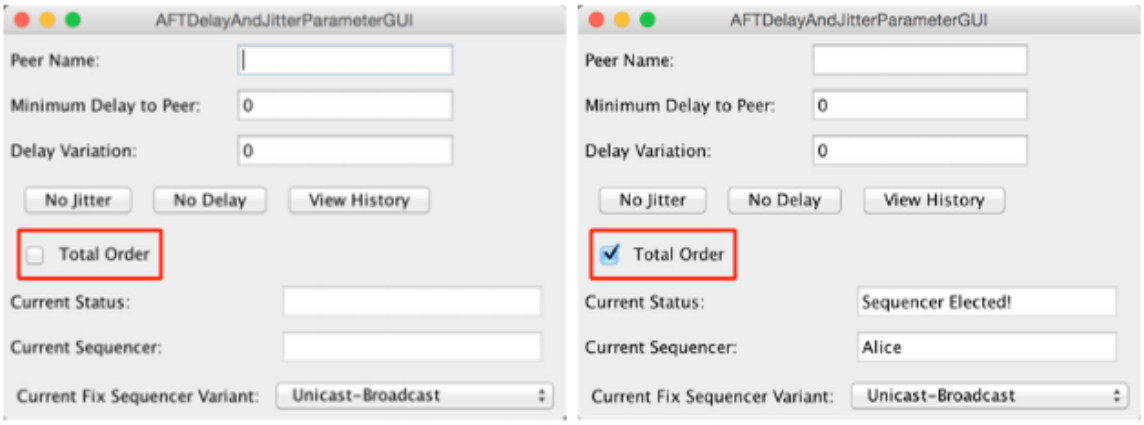


Fig. 19 Non-Total Order vs Total Order

If a user wants to send a message, the application will check with the FTManager whether total order broadcast is used (Fig. 20). If total order is not used, it will follow the regular process, which is adding the message into local model and notifying the other users about the change of the model. FTManager needs to keep a history of all the messages that is used for recovering when a new client joins a previous session. To keep the history updated, an extra step, adding the message to FTManager, is added to the regular process.

```
if (ftManager.isFT()) {
  ARegistry.sendRequest(communicator, listEdit, ftManager.getFTType());
} else {
  ListEditSent.newCase(communicator.getClientName(), listEdit.getOperationName(),
      listEdit.getIndex(), listEdit.getElement(), listEdit.getList(),
      AddressedSentMessageInfo.OTHERS, this);
  super.observableAdd(index, element);
  ftManager.addToHistory(listEdit);
  notifyAddReplicatingObservers(index, element);
  communicator.toOthers(listEdit);
}
```

Fig. 20 Send with/without total order

If total order is used, the static method sendRequest in ARegistry will be called. As I mentioned before, ARegistry is a link between communicator and FTManager. ARegistry will first look up the corresponding FTManager by giving a communicator then call the static method requestSending in FTManager to send a SentRequest which is wrapped by that FTManager it just looked up (Fig. 21). SentRequest (Fig. 22) contains a message, the user who sends the message and current variant of fixed sequencer algorithm. Then the requestSending method in FTManager will send SentRequest to either the sequencer or all destinations depending on the variant (Fig. 23) of fixed sequencer algorithm (Fig. 24). After receiving that SentRequest, the sequencer will call processReceivedSentRequest (Fig. 25 and Fig. 26) to process the received SentRequest. In BB variant, all destinations will also receive the SentRequest, they will buffer it and process it until they receive AMessageWithHashCode.

```
public static void sendRequest(Communicator communicator, ListEdit listEdit, String variant) {
  AFTManager ftManager = (AFTManager) getFTManager(communicator);
  AFTManager.requestSending(communicator, (ASentRequest) ftManager.warpSentRequest(listEdit));
}
```

Fig. 21 SendRequest in ARegistry

```
public interface SentRequest {
  String getClientName();

  ListEdit getListEdit();

  String getFTType();
}
```

Fig. 22 Interface of SendRequest

```
public interface FTType {

  static final String UB = "Unicast-Broadcast";

  static final String BB = "Broadcast-Broadcast";

  static final String UUB = "Unicast-Unicast-Broadcast";
}
```

Fig. 23 Variants of Fixed Sequencer Algorithm

```
public static void requestSending(Communicator communicator, ASentRequest sendRequest) {
  ListEdit listEdit = sendRequest.getListEdit();
  switch (sendRequest.getFTType()) {
    case FTType.UB:
      communicator.toClient(getSequencer(), sendRequest);
      SentRequestSent.newCase(communicator.getClientName(), OperationName.ADD,
          listEdit.getIndex(), listEdit.getElement(), listEdit.getList(), getSequencer(),
          AFTManager.class);
      break;
    case FTType.BB:
      communicator.toAll(sendRequest);
      SentRequestSent
          .newCase(communicator.getClientName(), OperationName.ADD, listEdit.getIndex(),
              listEdit.getElement(), listEdit.getList(), "ALL", AFTManager.class);
      break;
    case FTType.UUB:
      communicator.toClient(getSequencer(), sendRequest);
      SentRequestSent.newCase(communicator.getClientName(), OperationName.ADD,
          listEdit.getIndex(), listEdit.getElement(), listEdit.getList(), getSequencer(),
          AFTManager.class);
      break;
    default:
      break;
  }
}
```

Fig. 24 requestSending method in FTManager

```
public void objectReceived(Object message, String userName) {
  // need for integration with RPC
  if (message instanceof ASentRequest) {
    processReceivedSentRequest((ASentRequest) message, userName);
  } else if (message instanceof ListEdit) {
    processReceivedListEdit((ListEdit<E>) message, userName);
  } else if (message instanceof AMessageWithSeqNumFromSequencer) {
    processReceivedSeqNum((AMessageWithSeqNumFromSequencer) message, userName);
  } else if (message instanceof AMessageWithSeqNum) {
    processReceivedMsgWithSeqNum((AMessageWithSeqNum) message, userName);
  } else if (message instanceof AMessageWithHashCode) {
    processReceivedMsgWithHashCode((AMessageWithHashCode) message, userName);
  }
}
```

Fig. 25 ObjectReceived Method

When processing the SentRequest, FTManager will first extract the message inside of the SentRequest increment the sequence number, and wrap the sequence number into it. In UB variant, wrapMessage method (Fig. 27) will return a AMessageWithSeqNum, which only contains the message and the sequence number. In BB variant, the wrapMessage method (Fig. 27) will return a AMessageWithHashCode, which contains the message, the sequence number, and

the hashcode of the message. In UUB variant, the wrapMessage method (Fig. 27) will return a AMessageWithSeqNumFromSequencer, which also only contains the message and the sequence number. If we did not distinguish AMessageWithSeqNumFromSequncer from AMessageWithSeqNum, in UUB variant, when the user gets the message with sequence number back from the sequencer, it will deliver the message instead of broadcast it to all destinations. After wrapping the message, the sequencer will either send back a message to the sender or broadcast a message depending on the variant (Fig. 26). In UUB variant, the user will receive AMessageWithSeqNumFromSequencer after send the SendRequest. Upon receiving the AMessageWithSeqNumFromSequencer, the application will then broadcast that message to all destinations.

```
public synchronized void processSentRequest(ASentRequest<T> aSentRequest) {
  ListEdit<T> listEdit = ftManager.unwrapSentRequest(aSentRequest);
  if (aSentRequest.getFTType().equals(FTType.BB)) {
    ftManager.setAlgorithmStatus("Message buffered!");
    ftManager.bufferMessage(listEdit);
  }
  if (communicator.getClientName().equals(ftManager.getSequencer())) {
    MessageWithObj msg = ftManager.wrapMessage(listEdit, aSentRequest.getFTType());
    if (aSentRequest.getFTType().equals(FTType.UUB)) {
      AMessageWithSeqNumFromSequencer message = (AMessageWithSeqNumFromSequencer) msg;
      MessageWithSequencerNumberSent.newCase(communicator.getClientName(), OperationName.ADD,
          listEdit.getIndex(), listEdit.getElement(), tag, aSentRequest.getClientName(), this);
      communicator.toClient(aSentRequest.getClientName(), message);
      ftManager.setAlgorithmStatus("SeqNumFromSequencer Sent!");
    } else if (aSentRequest.getFTType().equals(FTType.UB)) {
      AMessageWithSeqNum message = (AMessageWithSeqNum) msg;
      MessageWithSequencerNumberSent.newCase(communicator.getClientName(), OperationName.ADD,
          listEdit.getIndex(), listEdit.getElement(), tag, AddressedSentMessageInfo.ALL, this);
      communicator.toAll(message);
      ftManager.setAlgorithmStatus("MsgWithSeqNum Sent!");
    } else {
      AMessageWithHashCode message = (AMessageWithHashCode) msg;
      MessageWithSequencerNumberSent.newCase(communicator.getClientName(), OperationName.ADD,
          listEdit.getIndex(), listEdit.getElement(), tag, AddressedSentMessageInfo.ALL, this);
      communicator.toAll(message);
      ftManager.setAlgorithmStatus("MsgWithHashCode Sent!");
    }
  }
}
```

Fig. 26 processSentRequest depends on the variant of fixed sequencer algorithm

```
public MessageWithObj wrapMessage(ListEdit listEdit, String variant) {
  incLocal();
  if (variant.equals(FTType.UB)) {
    return new AMessageWithSeqNum(localSeqNum, listEdit);
  } else if (variant.equals(FTType.BB)) {
    return new AMessageWithHashCode(localSeqNum, listEdit.toString().hashCode(), listEdit);
  } else if (variant.equals(FTType.UUB)) {
    return new AMessageWithSeqNumFromSequencer(localSeqNum, listEdit);
  }
  return null;
}
```

Fig 27. wrapMessage method

The final step in all destinations is to process the received object, which can categorized into two category, MessageWithSeqNum and MessageWithHashCode (Fig. 25). For MessageWithSeqNum, if the sequence number is not the one that the destination expected, destinations will buffer the message and process it later, otherwise, it will deliver the message.

For MessageWithHashCode, destinations will retrieve the corresponding message from the buffer and deliver it. As I previously stated, the delivered message will be added to FTManager to keep the history updated.

The history in FTManager will be used to recover the conversation when new clients join a previous session. When new clients join the session, they can recover the previous conversation by clicking the view history button. The recover method in sequencer will be triggered. It will send all previous messages one by one to new clients (Fig. 28).

```
public void recover(String aClientName) {
  if (communicator.getClientName().equals(getSequencer())) {
    for (ListEdit e : getHistory()) {
      communicator.toClient(aClientName, e);
    }
  }
  HistoryRecovered.newcase(aClientName, aClientName, this);
}
```

Fig. 28 Recover in FTManager

In distributed systems, processes may fail at any time. There is a chance that the sequencer could fail at any time. The library will call the leave method (Fig. 29) when clients crash or leave the current session and then FTManager will receive that which client left. If the client is the sequencer, all the process will start the re-election process (Fig. 30). The re-election process is considered to be two parts, backup sequencer take over the sequencer position and another client is chosen to be the backup sequencer.

```
this.addWindowListener(new WindowAdapter() {
  public void windowClosing(WindowEvent e) {
    communicator.leave();
    FTGUI.this.dispose();
  }
});
```

Fig. 29 Client left

```
public void reElect() {
  // TODO find a more elegant way to elect the sequencer
  gui.setAlgorithmStatus("Sequencer Re-electing!");
  sequencer = sequencerCandidate;
  for (String s : clients) {
    if (!s.equals(sequencerCandidate)) {
      sequencerCandidate = s;
      globalSeqNum = localSeqNum;
    }
  }
  gui.setSequencer(sequencer);
  gui.setAlgorithmStatus("Sequencer Re-elected!");
  SequencerReElected.newcase(sequencer, sequencer, this);
  if (isFT()) {
    gui.refresh();
  }
}
```

Fig. 30 Re-election process in FTManager

One of our goals is to separate the algorithm from the application. A complete traceable algorithm for the UB variant is illustrated as follows. BB and UUB variants currently share the same traceable algorithm with UB.

Client Join: When a client joins a session, join method is called by the library and "FTManager" (listener) will start the election process. If the current sequencer is blank, the user will be automatically chosen as the sequencer (SequencerElected). Since we also want to tolerate sequencer crash failure, a backup sequencer is also needed. When the second user joins the session, the backup sequencer is assigned to that user.

```
I***EvtType(trace.ft.SequencerElected) EvtSrc(ft.letao.AFTManager) Alice
```

If a user joins a previous session and there exists message history between other users, it can recover the history by clicking a button (HistoryRecovered).

```
I***EvtType(trace.ft.HistoryRecovered) EvtSrc(ft.letao.AFTManager) Bob
```

Send Messages: Whenever a user wants to send a message, it always wraps the message into a "SendRequest" and unicasts to the sequencer (SentRequestSent). Upon receiving the "SentRequest", sequencer fetches the message from the "SentRequest" and wraps it with sequencer number into "MessageWithSeqNum" and broadcast to all users (SentRequestReceive, MessageWithSeqNumSent) After receiving the "MessageWithSeqNum", the application either buffers the message or processes it bases on whether the sequence number is the next we expected (MessageWithSeqNumReceived).

Tracing from Bob:

```
I***EvtType(trace.ft.SentRequestSent) EvtSrc(ft.letao.AFTManager)
I***EvtType(trace.ft.MessageWithSequencerNumberReceived) EvtSrc(ft.
```

Tracing from Alice (sequencer):

```
I***EvtType(trace.ft.SentRequestReceived) EvtSrc(ft.AListInCoupler)
I***EvtType(trace.ft.MessageWithSequencerNumberSent) EvtSrc(ft.ARep
I***EvtType(trace.ft.MessageWithSequencerNumberReceived) EvtSrc(ft.
```

Client crash: When a client crash, a leave method is called by the application. If the client is the sequencer, a re-election process is needed (SequencerReElected).

```
I***EvtType(trace.ft.SequencerReElected) EvtSrc(ft.letao.AFTManager) Bob
```

4. System Illustration

4.1 Total Order of Message Delivery

Assume Alice, Bob, and Cathy are chatting with each other, but the latency between Bob and Alice is very high. Bob sends a message “Lunch?” to others. At the same time, Alice also sends a message “Done?” to the other. Not all the user will be consistent after all messages’ delivery (Fig. 16). If we enable total order broadcast, the messages will deliver in the same order for all users (App.1).

In the case below, Bob have 10 seconds delay to Alice. When Bob send “Done” to others while Alice send “Lunch?” to others, inconsistence arises.

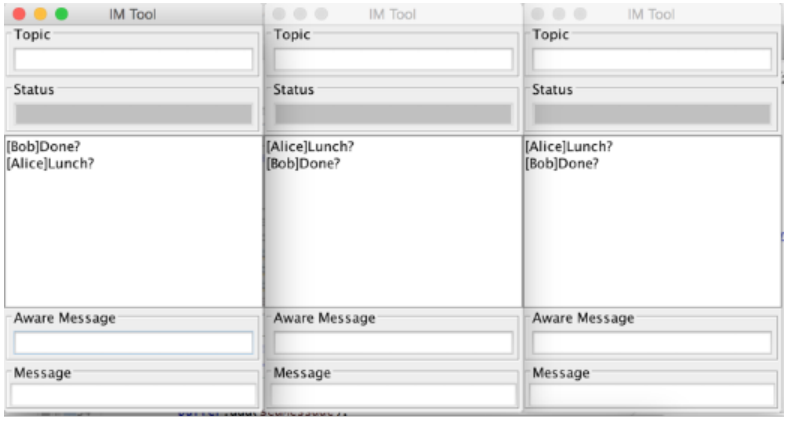
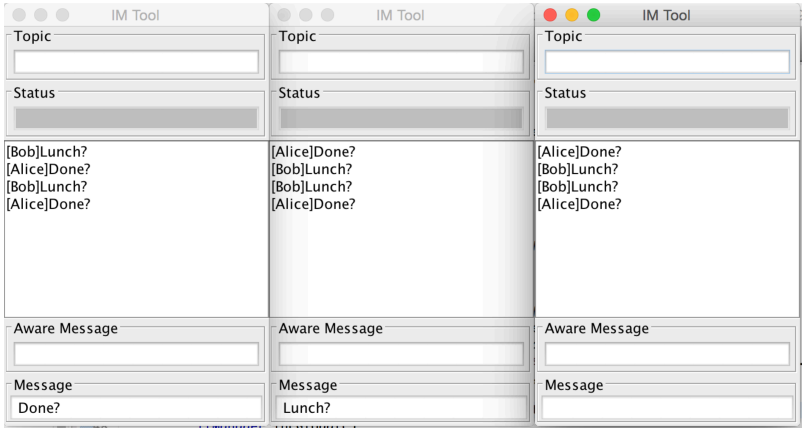


Fig. 16 Concurrent Message

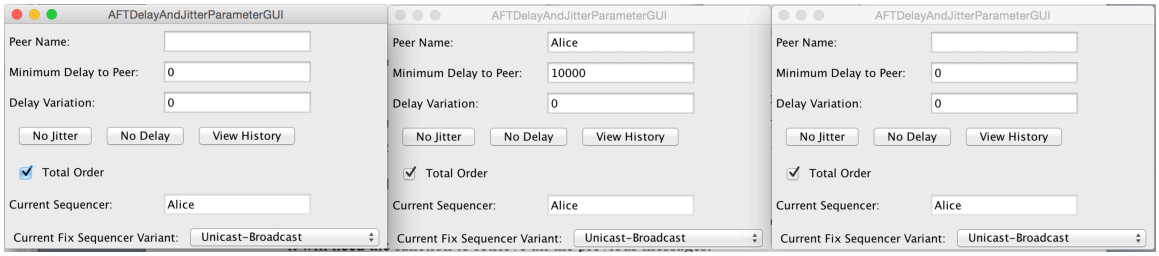
After we enable the total order broadcast, the messages will be delivered in the same order to all clients as shown below.



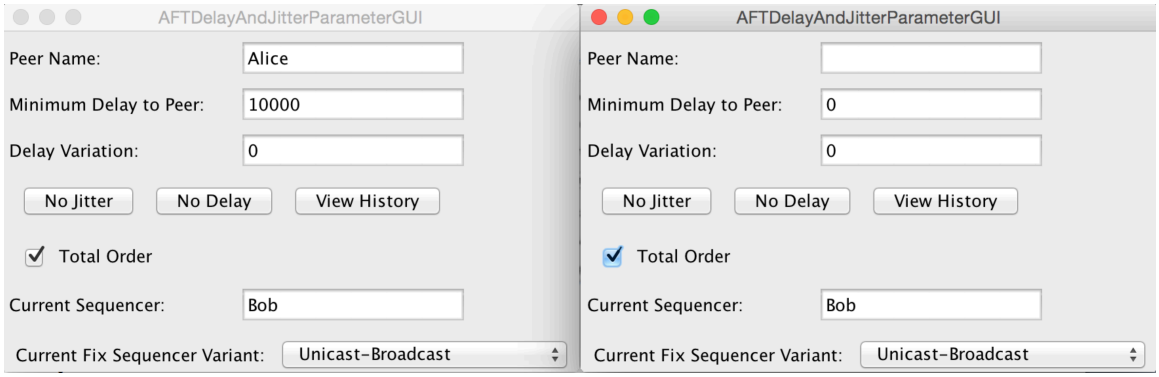
4.2 Re-Election of Sequencer

This functionality is used to tolerate crash failure on the sequencer process. It uses the technique of primary-backup. Suppose that Alice is now the sequencer but the application crashes. Bob will immediately take over the role of sequencer when he detects Alice's failure.

In the case below, there are three clients using the application and the current sequencer is Alice.



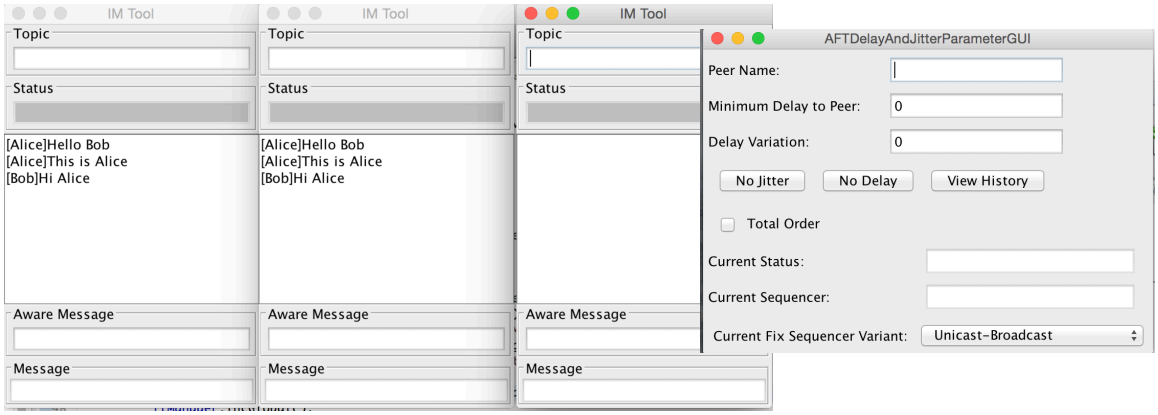
When the sequencer Alice crashes or left the session, the application will automatically re-elect the next sequencer. Bob now becomes the sequencer as shown below.



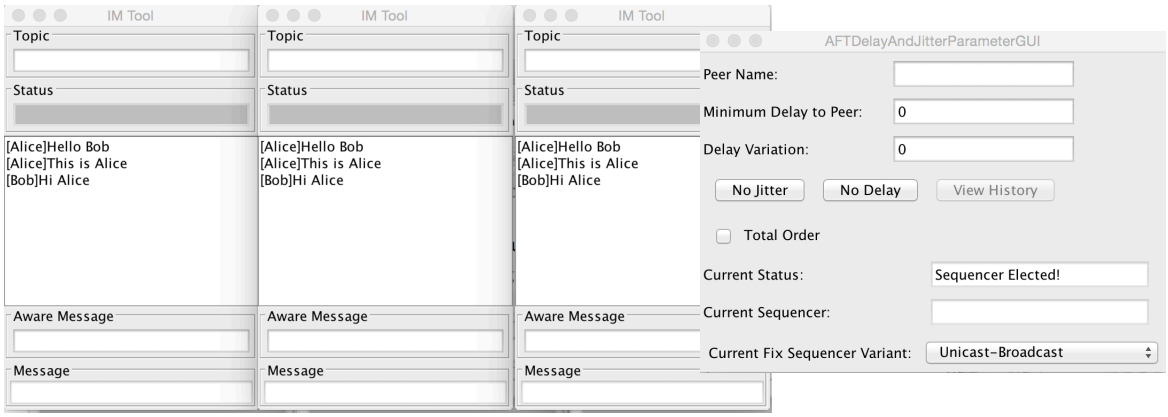
4.3 Recover History

This functionality is used when a new user joins the previous session or a crashed user comes back, it will need the function to retrieve all the previous messages.

In the case below, Alice and Bob are chatting and a new client Cathy joins the session.



By clicking the view history button, Cathy can get previous history as shown below.



5. Conclusions and Future Work

Fault-tolerant broadcast is one of the fundamental components to the building of any distributed systems. Many OS do not provide fault-tolerant abilities so it needs to be implemented at a higher level. Much research has been done in the filed of fault-tolerant broadcast and several algorithms have been proposed, but it's rare that they provided a visual demonstration.

Our distributed system provides a visual way to explore the fault-tolerant broadcast algorithm. Base on the fault-tolerant broadcast, the system has the ability to tolerate single process crash failure. We also created a shared library for the fixed sequencer algorithm so that the different variants of the algorithm can easily switch from one to the other.

Although the application can provide the basic functionalities described above, there are several areas for potential improvements.

Our application used the fault-tolerant technique, called primary-backup. Currently, the client list is sorted and the backup sequencer is the client whose name is after the sequencer and it seems unfair for the rest of the clients. A random election method can make the process much fairer.

The current application can use the three different variants of the same fixed algorithm, but users cannot see the differences among those variants. One can add constraints into the application to see the limitations of each variant. For instance, using a bounded buffer in the receiving thread of the sequencer to limit the number of messages can be sent every minute. UUB will behave as the most efficient variants since it is not depending on the sending speed of the sequencer.

In current implementation of BB variant, we still send the whole message body to all destinations, since java internal hashing function is object-dependent. The function will compute out different value for the same string in the different process. One approach is to use the SHA1 hashing function to compute hash code for strings. It can guarantee that the same string will get the same hash code in the different processes. Another approach is to use the host id of the sender and the number of message sent from the sender as a pair instead of using hashcode of messages.

References:

[1] Fault Tolerance. 5 Apr. 2015. <https://www.cs.rutgers.edu/~pxk/rutgers/notes/content/ft.html>.

[2] Bartlett, J., and J. Gray. *Fault Tolerance in Tandem Computer Systems*. S.l.: [s.n.]. Print.

[3] Schneider, Fred B., David Gries, and Richard D. Schlichting. "Fault-tolerant Broadcasts." *Science of Computer Programming*: 1-15. Print.

[4] V. Hadzilacos, and S. Toueg, "*A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*", Technical Report, Department of Computer Science, University of Toronto.

[5] X. DEFAGO, A. SCHIPER, and P. URBAN, "*Total Ordered Broadcast and Multicast Algorithms: Taxonomy and survey*", ACM Computing Surveys, Vol. 36, No. 4, December 2004, pp. 372–421.

[6] NAVARATNAM, S., CHANSON, S. T., AND NEUFELD, G. W. 1988. "*Reliable group communication in distributed systems*", In Proceedings of 8th IEEE International Conference on Distributed Computing Systems (ICDCS-8) (San Jose, CA). IEEE Computer Society Press. 439–446.

[7] Schiper, André, Kenneth Birman, and Pat Stephenson. "Lightweight Causal and Atomic Group Multicast." *ACM Transactions on Computer Systems* (1991): 272-314. Print.

[8] Gray, J., "*Why Do Computers Stop and What Can Be Done About It?*", Tandem TR85.7, June 1985, Tandem Computers, Cupertino, CA.

[9] Tanenbaum, Andrew S., and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. 2nd ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2007. Print.

[10] Moon, Todd K. (2005). *Error Correction Coding*. New Jersey: John Wiley & Sons. ISBN 978-0-471-64800-0.

[11] Burbeck, Steve (1992) *Applications Programming in Smalltalk-80(TM): How to use Model–View–Controller (MVC)*

[12] "Simple Example of MVC (Model View Controller) Design Pattern for Abstraction." - *CodeProject*. Web. 16 Apr. 2015. <http://www.codeproject.com/Articles/25057/Simple-Example-of-MVC-Model-View-Controller-Design>.

[13] Défago, Xavier, André Schiper, and Péter Urbán. *Totally ordered broadcast and multicast algorithms: A comprehensive survey*. No. LSR-REPORT-2000-012. 2000.

[14] P. Dewan, *Distributed Systems Course*, The University of North Carolina at Chapel Hill, Fall 2011.

[15] Dewan, P., "Programmer-controlled application-level multicast," *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2014 International Conference on* , vol., no., pp.332,341, 22-25 Oct. 2014

[16] Birman, Kenneth P. Reliable Distributed Systems Technologies, Web Services, and Applications. New York: Springer, 2005. Print.

[17] Hipschman, William. "*Semantics and Implementation of Broadcast.*" Print.

Appendix:

Video Links:

1. Total Order of Message Delivery: http://youtu.be/FX3H2FXzLxA
2. Sequencer Re-election: https://youtu.be/e3fEbeUbonM
3. History Recover: http://youtu.be/d3QjilIRAlY
4. Multicast Library: http://goo.gl/tsCvM3
5. Traceable Algorithm: http://goo.gl/cJBhfW
6. GitHub: https://github.com/letaoj/comp790
7. Distributed collaborative system course:
http://www.cs.unc.edu/~dewan/790-063/current/index.html
8. IM Application: http://goo.gl/gBRH4X

Algorithm pseudocode:

```
1:Sender:
2:    procedure TO-multicast(m)
3:        send (m) to sequencer
4:    end

5:Sequencer:
6:    Initialization:
7:        seqnum ← 1
8:    when receive (m)
9:        sn(m) ← seqnum
10:       send (m; sn(m)) to all
11:       seqnum ← seqnum + 1
12:   end when

13:Destinations (code of process p_i):
14:   Initialization:
15:       nextdeliver_pi ← 1
16:       pending_pi ← Ø;
17:   when receive (m; seqnum)
18:       pending_pi ← pending_pi ∪{(m; seqnum)}
19:       while ∃(m' ; seqnum' ) ∈ pending_pi : seqnum' = nextdeliver_pi do
20:           deliver (m')
21:           nextdeliver_pi ← nextdeliver_pi + 1
22:       end while
23:   end when
```

Fixed Sequencer Algorithm UB variant [13]

1:Sender:
2: **procedure** TO-multicast(m)
3: send (m) to all
4: **end**

5:Sequencer:
6: Initialization:
7: seqnum ← 1
8: **when** receive (m)
9: sn(m) ← seqnum
10: send (hash(m); sn(m)) to all
11: seqnum ← seqnum + 1
12: **end when**

13:Destinations (code of process p_i):
14: Initialization:
15: $nextdeliver_{pi}$ ← 1
16: $buffer_{pi}$ ← Ø;
17: $pending_{pi}$ ← Ø;
18: **when** receive (m)
19: $buffer_{pi}$ ← $buffer_{pi}$ ∪{(hash(m), m)}
20: **when** receive (hash(m); seqnum)
21: m ← $buffer_{pi}$ get(hash(m))
22: $pending_{pi}$ ← $pending_{pi}$ ∪{(m; seqnum)}
23: **while** ∃(m' ; seqnum') ∈ $pending_{pi}$: seqnum' = $nextdeliver_{pi}$ **do**
24: deliver (m')
25: $nextdeliver_{pi}$ ← $nextdeliver_{pi}$ + 1
26: **end while**
27: **end when**

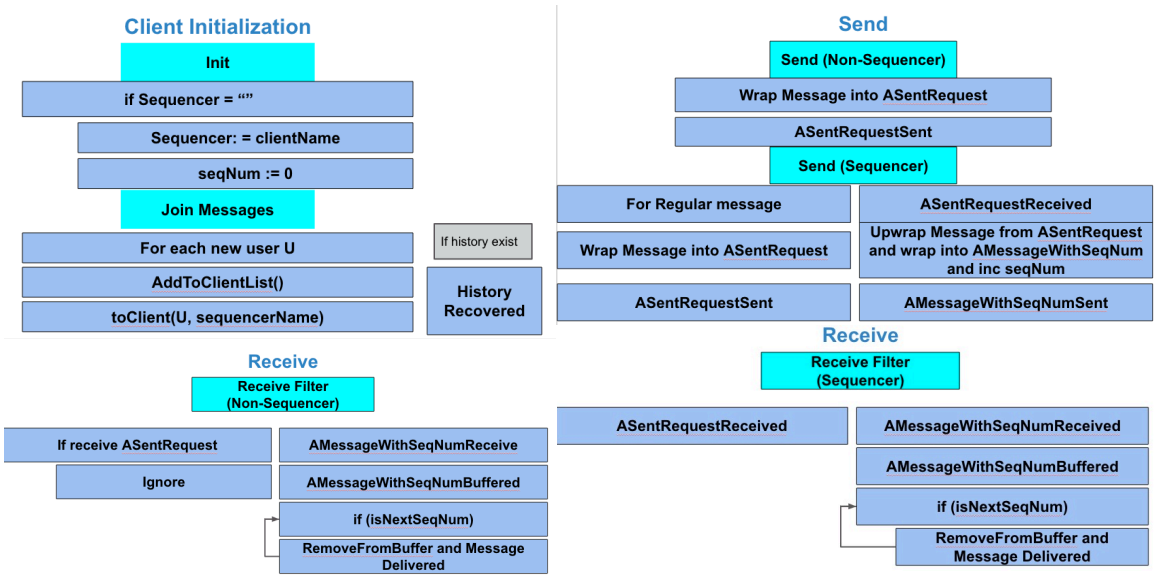
Fixed Sequencer Algorithm BB variant

```
1:Sender:
2:    procedure TO-multicast(m)
3:          send (m) to sequencer
4:          wait for seqnum
5:          when receive (seqnum)
6:          send (m; seqnum) to all
7:    end

8:Sequencer:
9:    Initialization:
10:         seqnum ← 1
11:   when receive (m)
12:         sn(m) ← seqnum
13:         send (m; sn(m)) to caller
14:         seqnum ← seqnum + 1
15:   end when

16:Destinations (code of process p_i):
17:   Initialization:
18:         nextdeliver_pi ← 1
19:         pending_pi ← Ø;
20:   when receive (m; seqnum)
21:         pending_pi ← pending_pi ∪{(m; seqnum)}
22:         while ∃(m' ; seqnum' ) ∈ pending_pi : seqnum' = nextdeliver_pi do
23:               deliver (m')
21:               nextdeliver_pi ← nextdeliver_pi + 1
22:         end while
23:   end when
```

Fixed Sequencer Algorithm UUB variant



Completely Traceable algorithm for UB variant