Fault-Tolerance in Distributed Systems

*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 the 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 system 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. 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 system has change the entire world as well as people’s daily life. When people use the web browser and try to connect to a web server, it is participating in what seems to be a simple form of a client/server distributed system. When people use the web application provided by modern server such as Google or Facebook, they are not just interacts with one single machine, however; these applications are built from a large collection of machines. It is possible that machines may fail at any time, and it is not the scene that people want to see when one machine failure that brings the entire application down. Fault tolerance becomes one of the central challenges in constructing distributed systems. Many Researches have been done on fault-tolerance in distributed system but people except the researchers may not have a direct view of the algorithm. A distributed system is needed to illustrate the fault tolerance algorithm.

The precondition of a fault tolerance distributed system is the atomic broadcast. A well-designed atomic broadcast mechanism will bring the benefit of fault tolerance and will increase the likelihood of building a fault-tolerant distributed system. The first step we want to achieve is to implement the fault-tolerant broadcast in our application. We want to implement this functionality in a modular fashion means one can either enable or disable the fault-tolerant broadcast module.

We want to create a shared library for the fixed sequencer algorithm to exploit the commonality of the variants of the algorithm. The fixed sequencer algorithm can have three variants, but each variant is similar to the other. A shared library can have some common code that every variant can use it.

We also want to create traceable algorithm in the application to separate the algorithm and code. Finally, we will use the application to demonstrate the above goals visually.

**2. Background**

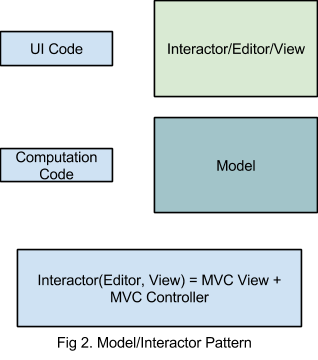
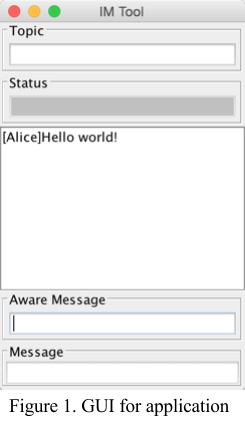
**2.1 Fault tolerance**

The phrase “*fault-tolerance*” is a combination of two single words, “fault” and “tolerance”. “Fault”, in the system point of view, can be defined as a malfunction or deviation from expected behavior. The meaning of “tolerance” is the ability to allow 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, it allows the computer keep executing with the presence of defects. The general approach to building fault tolerant system is redundancy [9]. There are three different kinds of way: 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 occurs. 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.

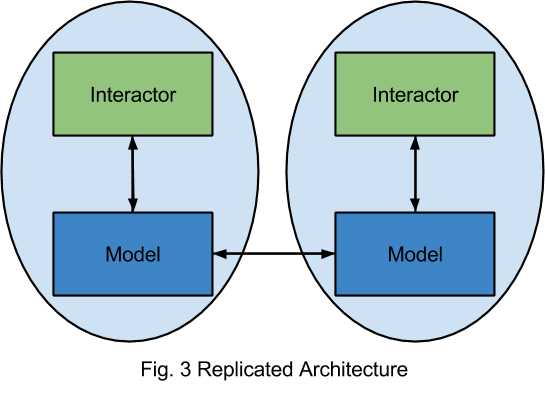
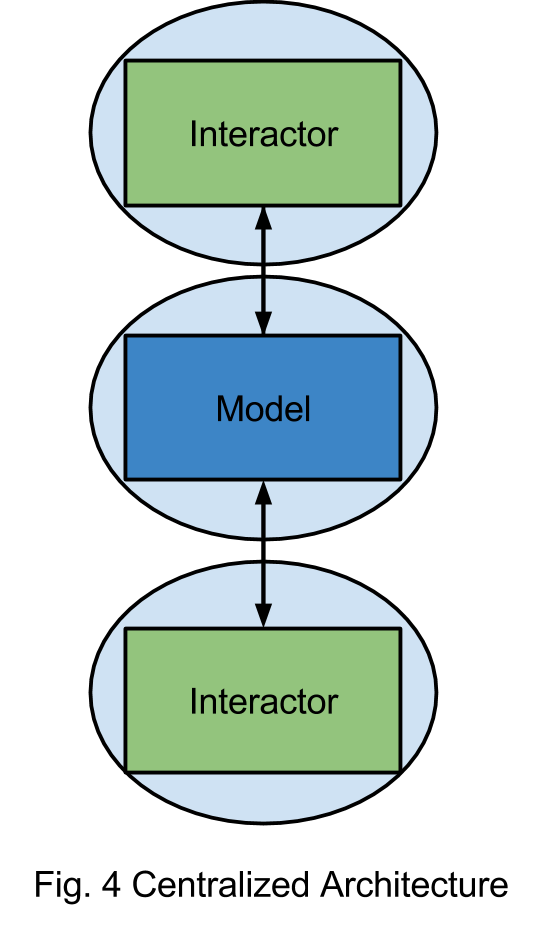
One example of well-known system that combines the fault tolerant techniques is called Tandem Computer System. Systems whose fault tolerance is based solely on hardware mechanisms can be designed to provide high reliability and continue to operate in the presence of hardware component failure, however,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

Each component in the Tandem system is designed to tolerate faults. For example, 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 fails. Most faults in production software are transient [8] so when primary process fails, programs can continue execution in the backup process when the software bug is transient.

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

**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 system. The first one is that a distributed system consists of components (i.e., computes) that are autonomous and the second one is that each single user thinks they are dealing with single systems [9]. One example of distributed system 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 their use the system. By join 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 user can type the words. When a user typing in the aware message text field, other users in the same session will notice that others are typing by seeing the display text in the status. The following link is a demo of the application: <http://goo.gl/gBRH4X>.

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 in my application. A String list is used as the model of the application. We combine the view and controller into the GUI and let GUI can take input or display output of the model. We call the combination of the view and controller an interactor.

Since the application is a distributed system, we will need some mechanism to distribute the model to all users. There are two way to share a model, replicated and centralized model sharing. Figure 3 and Figure 4 illustrates the two different architectures of the model sharing. The two type 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 that the application is an IM application, and we don’t want to wait for a long time to get local feedback. An efficient broadcast mechanism can solve the issue of consistency caused by causality and concurrent operations.

**2.3 Failure Type**

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

It is important to characterize the failures when one is try 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 fails to respond to incoming requests.
  + *Receive omission*. A process fails to receive incoming messages.
  + *Send omission*. A process fails to send 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 is a process can receive or send message until time t, when a crash failure occurs in the process.

A process can undergo two different omission failures, send omission and receive omission. A process in send omission cannot send messages to the other processes but can still receive messages and perform the action it receives. A receiving omission is the converse. A general omission failure exhibits send omission failures, receive omission, or both [1, 4, 9].

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

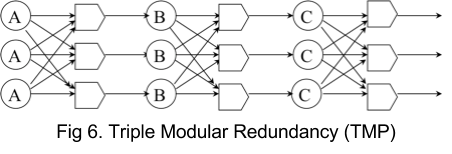
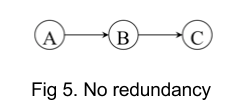
*Arbitrary failures* are the most serious type of failure, and also known as Byzantine failures. In our example, if ‘A’ process undergo the arbitrary failures and wants to send a “lunch?” to the other processes, then it may send “done?” to other processes instead [9].

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

**2.4 Approach Fault tolerance in distributed system**

The general approach to achieve fault tolerance in distributed system 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, if we consider a system that 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. The TMR design, all the component 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’ halt or one of the processes ‘A’ produce a random output, the input goes to ‘B’ will always be the correct one. The three 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.

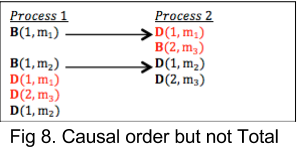
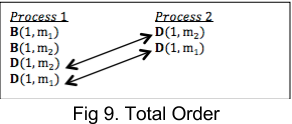
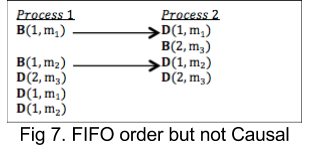
With a *Primary backup* approach, one process (the primary) does all the work. When that process fails, the backup takes over. We used this approach to tolerant the sequencer process failure in our implementation.

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 message to one process at a time. If a process p wishes send a message m to other people in the network, it will need to send the message separately. 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 the reliable broadcast and it do not ensure any orders. Reliable broadcast only ensures that all correct processes will eventually deliver a message that broadcasted by a correct process. Stronger variants of reliable broadcast will force more requirements on the order of deliver of the messages. [3]

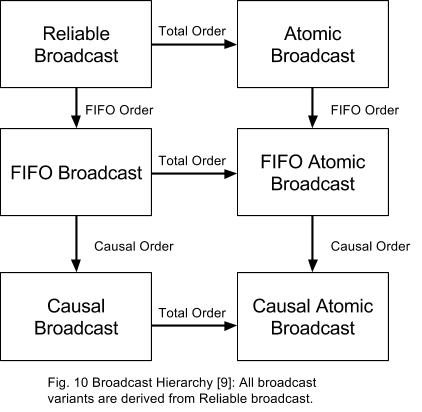
There are three different order of messages, *FIFO order*, *Causal order*,and *Total order*.

*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 m1 before message m2, then no correct process delivers m2 before m1. (Figure 7)

*Causal order.* If a correct process broadcast a message m1 causally precedes the broadcast of a message m2, and then no correct process delivers m2 before m1. (Figure 8)

*Total order.* If correct processes p and q both deliver message m1 and m2, then p delivers m1 before m2 if and only if q delivers m1 before m2. (Figure 9)

** By forcing the deliver order of the message to the reliable broadcast, three variants of reliable broadcast can be proposed, FIFO reliable broadcast, Causal reliable broadcast, and Atomic reliable broadcast. Figure 10 shows how all the broadcast variants can be derived from the basic broadcast type – reliable broadcast.

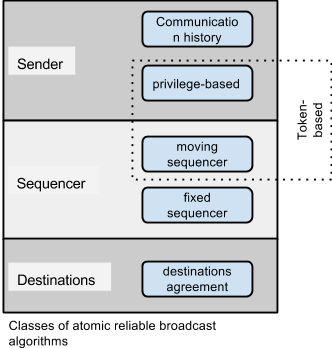
FIFO reliable broadcast, one guarantee the FIFO order of the deliver message and always supported in the TCP/IP protocol.

Causal reliable broadcast is another variant of reliable broadcast, which strengthen the FIFO broadcast. Sometimes FIFO order is not enough because FIFO ordering too limited in context (a single process). For example, User ‘A’ send a paper to User ‘B’ and ‘C’, User ‘B’ send another paper that can be understood after reading the paper sent by ‘A’ first. User ‘C’ get the paper sent by ‘B’ first and reply with his understand 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 the causal order to the system, it could solve the above problem. The paper sent by User ‘B’ will never deliver to User ‘C’ until the paper sent from User ‘A’ deliver to User ‘C’.

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 m1a “remove paragraph 1”. Process b broadcast message m1b “remove paragraph 2”. If the messages deliver to the first article in the order of m1a, m1b, and deliver to the second article in the order of m1b, m1a, 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 guarantee the above property.

**2.6 Atomic Broadcast algorithm**

There are different algorithms of Atomic reliable broadcast. Before introduce the algorithm, we will first define the notation will be used in the algorithm. A sender process is process ps from which a message is sent. A destination process is a process pd 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 take several roles simultaneously, however, the meaning of each role is totally different from each other.

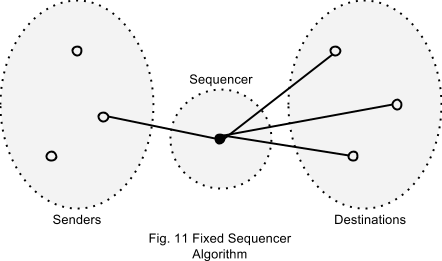
Following from the three different roles that processes can take, atomic broadcast can fall into three basic classes, depending where the order of message is built [13]. This allows us to define subclasses as illustrated in the above figure. This results in the five classes illustrated on the figure: *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 the sequencer crashes. Figure 11 illustrates the mechanism of fixed sequencer algorithm. The solid black circle in the figure represents the sequencer.

The pseudo-code illustrates the one of the approach to the algorithm. One certain process takes over the role of sequencer and builds the total order [13]. This approach did not have the fault tolerance mechanism in it so it cannot tolerate the failure of the sequencer.

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 warps it into *m* to the destinations. The destinations then deliver messages in sequence thus according to the total order built by the sequencer.

**

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 pi):

14: Initialization:

15: nextdeliverpi 🡨 1

16: pendingpi 🡨 Ø;

17: **when** receive (m; seqnum)

18: pendingpi 🡨 pendingpi ∪{(m; seqnum)}

19: **while** ∃(m’ ; seqnum’ ) ∈ pendingpi : seqnum’ = nextdeliverpi **do**

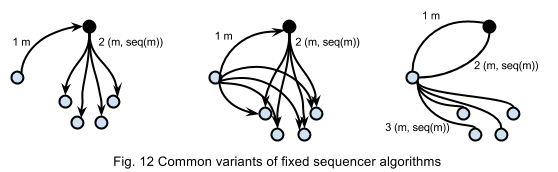
20: deliver (m’)

21: nextdeliverpi 🡨 nextdeliverpi + 1

22: **end while**

23: **end when**

Fig 12. Simple fixed Sequencer Algorithm [13]

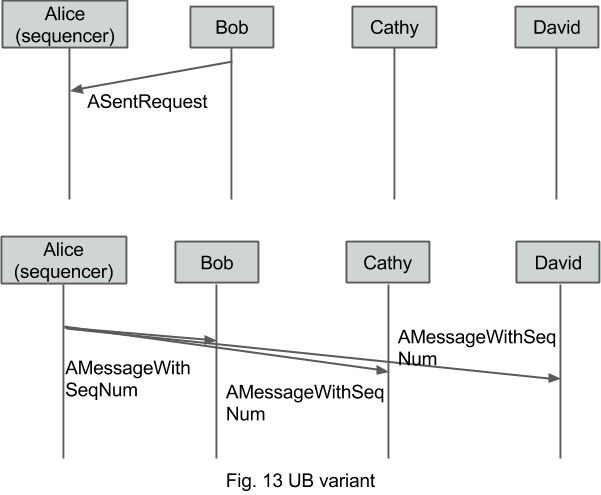
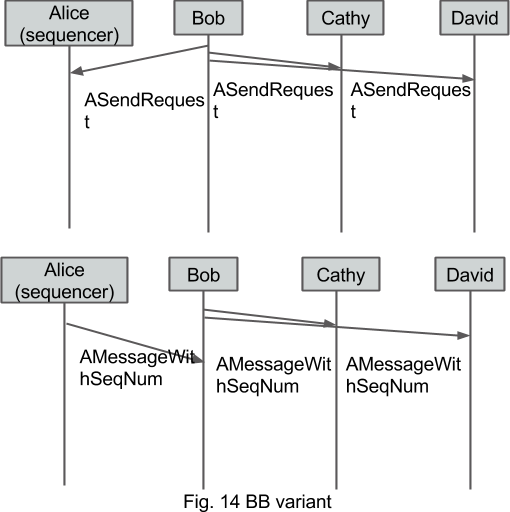
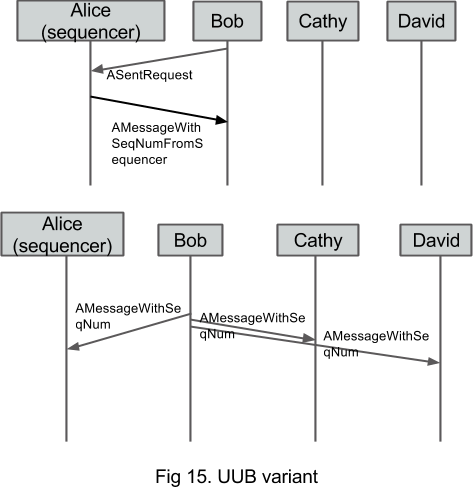
The fixed sequencer algorithm can have three different variants. Figure 12 illustrates some common variants of fixed sequencer algorithm. (‘U’ stands for unicast and ‘B’ means broadcast.) Then those variants of fixed sequencer algorithms can be easily understood. The first variant “UB”(“Unicast-Broadcast”) consists of a unicast to the sequencer, followed by a broadcast by sequencer, which is the simple 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 reply with a unicast to the sender followed by a broadcast to all destinations from the sender.

**2.8 Multicast Library**

A shared library typically means the same pieces of code that can be reused by different programs or application. My application is built on the top of the multicast library created by Professor Dewan at UNC-Chapel Hill. The functions provided in the library simplify the step of building my distributed systems. For example, it provides the remote process communication method so that I don’t need to build up my application from nothing and handle the communication issue between processes. Another important function that provided by the library and I used in my application is the message filter. By creating different message filters, one can add information to a sending message or subtract information from the receiving message. A shared library can simplify the building process as well as reduce the length of the code [14]. Refer to video and paper.

**2.9 Traceable Algorithm**

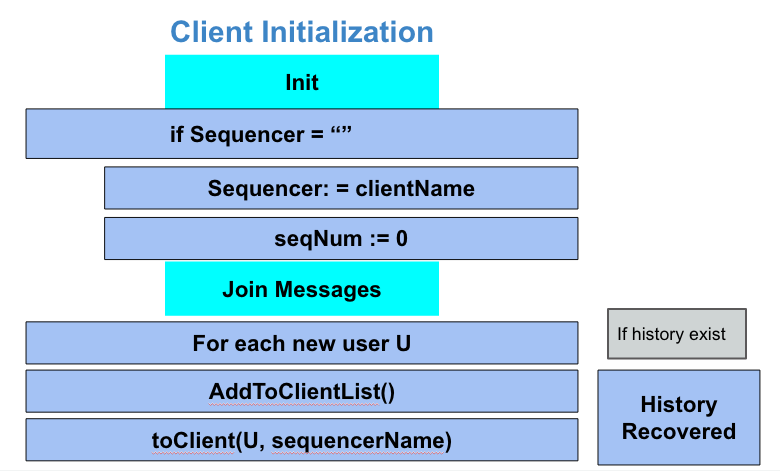
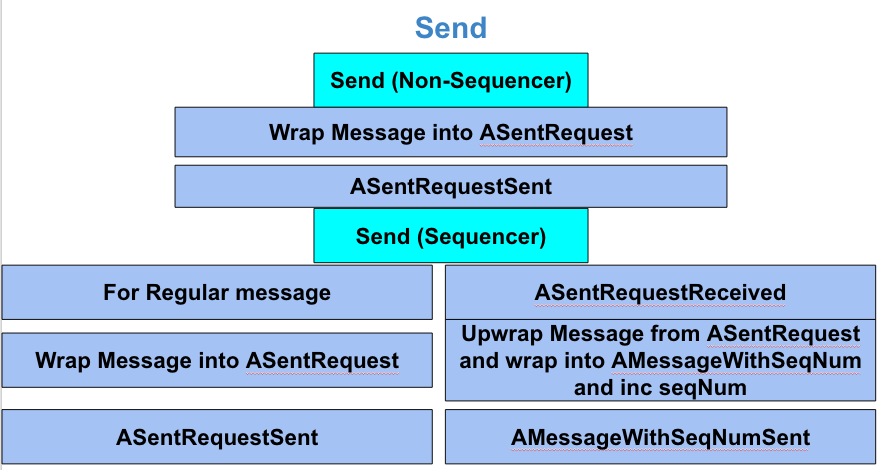
An algorithm is needed for understanding a program, testing and debugging but in most of the case, algorithm is tightly tied to the program. Using in-line algorithm can extract comments from code to view algorithm but it do not get a linear path from scattered object. We can get the linear path by adding 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, breakpoint cannot be transferred to another computer. Traceable algorithm provides an elegant way to separate the algorithm from the program. It provides an executable specification of algorithm implemented with code. It allows algorithm steps to be in separate packages, filtering by event type, and ways to find implementations of a step [14]. Refer to video.

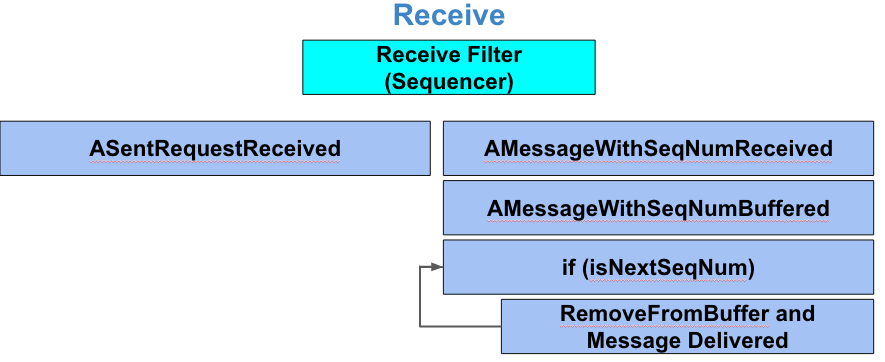
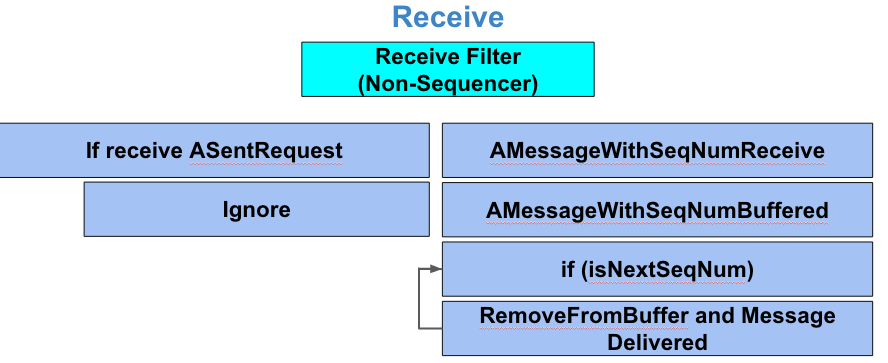
**5. My Work**

We first applied the fixed sequencer algorithm into our distributed system, and let IM application can broadcast messages in total order. Figure 13, 14 and 15 illustrate the modified fixed sequencer algorithm in our application. As the UB variant of the algorithm describes, whenever a user wants to send a message, the application will first wrap the message into a “SentRequest” and unicast this “SentRequet” to sequencer. Upon receive the “SentRequest”, sequencer will fetch the message in it and warp the message with a sequence number then broadcast to all the users.

An important component in the application is call “FTManager” that is in charge of the election of the sequencer, managing the order of the message, recovering the history when new client join, and the re-election of the sequencer. In the “FTManager” of the sequencer, it will need a “globalSeqNum” and “localSeqNum” since it plays the role of a sequencer as well as a user. The simple fixed sequencer algorithm cannot handle the case when the sequencer crashes. We use the primary-backup technique and make a slight modification to the fixed sequencer algorithm so that it can tolerate the crash fault on the sequencer process. When a sequencer is elected, a sequencer candidate will also be elected play the role of backup sequencer. The sequencer candidate will be active when the sequencer crashes.

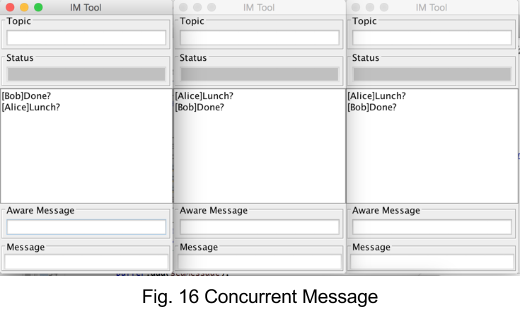
A shared library is created in the application for different variants of the fixed sequencer algorithm. By using the shared library, about 10 lines of code in total is needed to adjust one variant to another. The “SentRequest” have a data field to record the variant of the fixed sequencer algorithm, the sequencer will made the corresponding action by checking the type inside of the “SendRequest”.

A complete traceable algorithm for the UB variant is listed in the figures below.



**6. Demo**

**6.1 Scenarios**

Assume Alice, Bob, and Cathy are chatting to each other, but the latency between Bob and Alice is very high, when Bob send a message to the other, at the mean time, Alice also send a message to the other. The inconsistency will arise. (Fig. 16)

YouTube Link: [here](http://youtu.be/Hyyi5gdNUGE)

**6.2 Re-Election of Sequencer**

This functionality is used to tolerate the crash failure on the sequencer process, it use the technique of primary-backup. The total order still maintain after the re-election.

YouTube Link: [here](https://youtu.be/e3fEbeUbonM)

**6.3 Recover History**

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

YouTube Link: [here](http://youtu.be/d3QjilIRAlY)

**6.4 Traceable Algorithm**

This video is the traceable algorithm displays in the console while the program is running.

YouTube Link: [here](http://youtu.be/I0pBmK8iLOU)

**7. Conclusion**

Fault-tolerance broadcast is one of the fundamental components to the building of any distributed systems. Many OS does not provide the fault-tolerance abilities so it needs to be implemented in a higher level. Much research has been done on filed of fault-tolerance broadcast and a lot of algorithms have been proposed but rare of them provide a visually demonstration.

In our distributed system, it provides a visual way to explore the fault tolerance broadcast algorithm. Base on the fault-tolerant broadcast, the system have the ability to tolerate single process crash failure. We also create a shared library for the fixed sequencer algorithm that the different variant of the algorithm can easily switch from one to the other.

**8. Future Work**

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

Our application used the fault-tolerance 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 more elegant way to find the backup sequencer can be used instead of the current one; for example, a random election method can make the process much fairer.

The current application can used the three different variants of the same fixed algorithm, but users cannot see the differences among those variants. One can add constrains into the application to see the limitations of each variant. For example, 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 depend on the sending speed of the sequencer.

**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.