CS425 MP1: Design Document

Bhopesh Bassi, Zhongshen Zeng

February 28, 2017

Contents

1 Overview

In this document, we describe the design of a totally ordered fault tolerant chat room. We start by discussing the various protocols considered for each component of the application and explaining why we chose the one we did. In the second half of the document, we give an overview of the implementation details of the application and present some data to evaluate the performance.

2 Various tasks and the protocols considered

2.1 Multicast

When we started implementing the multicast, only reliable multicast protocol known to us was R-multicast studied in class. Even though the bandwidth was quadratic, we didn't have another choice at the time. It was only during its implementation and testing that we discovered gossip protocol from Professor Gupta's slides. Unfortunately, we already had some progress on R-multicast and in the interest of time didn't implement gossip based multicast.

2.2 Failure detection

Following protocols were considered for failure detection:

a. **Centralized heartbeating:** This protocol was discarded because it had a single point of failure. Making it failure tolerant required having redundant central servers or having a leader election protocol running in the system.

b. **Ring heartbeating:** Reasons to discard this protocol were two-fold. Firstly, it can only tolerate atmost k-failures, where k is the number of successors a process sends message to. Secondly, the overhead to repair the ring in case of failures seemed too much, especially when we had better protocols.

c. **All to All heartbeating:** Quadratic bandwidth requirements was the major reason to discard this protocol.

d. **Gossip:** Reasons for discarding this protocol were mainly based on the mathematical analysis(found in Professor Gupta's slides) that it is not the optimal protocol as it mixes failure detection and dissemination components. Moreover, each process periodically gossips entire membership list, so in a large datacentre where there are many processes, size of each gossip message would be very big.

e. **SWIM:** SWIM stands for Scalable Weakly-consistent Infection-style process group Membership protocol. This is the protocol that we chose and implemented. It uses ping-ack instead of heartbeating. Mathematical analysis show that detection time is constant when number of processes are large. Moreover, dissemination is piggybacked on top of ping-ack messages, so there is no extra bandwidth used for disseminating detected failures. Mathematical analysis show that it takes $O(logN)$ protocol periods for failure detection to reach the entire group.

2.2.1 SWIM failure detector

Implementation The original SWIM protocol has two kind of ping-ack messages, direct and indirect. Indirect pings are used when a process fails to reply to a direct ping. In our original implementation (SWIMFailureDetector.java), we were sending indirect pings each on a separate thread. When we tested this implementation with simulated delays in the network, we discovered that in case of delays, direct pings almost always fail and when threads are spawned for indirect messages, they take longer than protocol period to reach intended process. We suspect that this happens because the VMs are single-core and when delay is added in network at every point to point contact, threads get switched out by the OS, many process wait for I/O and then compete for CPU. All of this adds a lot of delay to the receipt of indirect messages.

As a solution to this problem, we decided to eliminate indirect messages in the version2 implementation (SWIMFailureDetectorV2.java) of the protocol and only picked k-random processes for direct pings. These k pings are sent sequentially one after the other each protocol period instead of parallely on different threads.

Mathematical properties

a. The first detection time of the protocol is constant: $\frac{e}{e-1}$ protocol periods. This result is true for very large number of processes i.e when N reaches infinity. This isn't true for our case but nevertheless, first detection time is still low.

b. False positive rate is tunable by number of indirect pings in version1 implementation. It is not tunable in version2 implementation but as explained already indirect pings were creating problems with network delays on single core machine. We can tune false positive rate of version2 implementation by changing the timeout of ping message.

c. Completeness is achieved with $O(logN)$ protocol periods because failure dissemination is infection style dissemination.

d. Bandwidth used by each process per protocol period is constant as each process pings k other processes. k is independent of n.

Parameters for V2 implementation

a. Ping timeout: Time for which pinging process should wait for acknowledgement before declaring pinged process as having failed. Without any packet loss and with one sided maximum network delay of 100 ms, a timeout of 600 ms was working fine and we were not getting false positives. However when we tested with one way network delay of 100 ms and packet loss of 4%, false positive rate suddenly increased. As we are using TCP, messages were not getting dropped but TCP had to send some messages many times. Debugging showed that in some of the cases, time elapsed between ping send on one process and ping receive on another process was as high as 4000 ms. Due to this, we have set a timeout of 5000 ms now, it seems to work but because we had no good way of calculating the maximum possible delay because of packet loss, **it is possible that we see some false positives during the demo.** This is also making our detection time go higher in normal case when there are no network delays but there's no better way here. Its just a trade off between detection time and false positives. We tried to eliminate false positives totally under the simulated network conditions suggested on Piazza.

b. Number of ping targets: Number of processes to be pinged each protocol period. Higher the value, more quickly we discover failures but higher the bandwidth used too.

c. Protocol period: Length of one protocol period. This should be set such that we can send ping to k processes sequentially and also have some additional time. Set to 17000 ms for now(due to 4% packet loss)

d. Minimum protocol period: Fraction of the total protocol period, that must be elapsed each round. If we receive acks before 5000 ms, we don't want to wait for entire protocol period to run next round. This will cause detection time to be very high. Value of this parameter is set to be 0.06 effectively making each protocol period of 1000 ms when there are no delays on network.

e. Lambda(λ): We piggyback the failures detected in last $\lambda logN$ protocol periods on top of each ping and ack message. Larger the value of lamda, more we can be sure of disseminating failure information to everyone in the group.

2.3 Network protocol

We considered TCP and UDP. Ideally, we should have used TCP for multicast because network is unreliable and UDP for failure detection because we have the liberty of trading reliability for lesser resources. As we had to have TCP in the system, we decided to use TCP for both and only later if time allowed, switch failure detector to UDP. We don't have time left to make the switch now but as explained in section 4, the failure detection(and multicast) code is totally abstracted from the network protocol and we can switch protocols at any time. It's just a matter of writing new implementations of couple of interfaces.

2.4 Total ordering

We considered both algorithms studied in the class for total ordering:

a. **Sequencer based:** This algorithm was discarded for the reasons similar to centralized heartbeating for failure detection. We didn't want to have single point of failure, neither did we want to have redundant sequencers and/or leader election protocol running in the system.

b. **ISIS algorithm:** This algorithm seemed a good choice because it was decentralized. We use point to point TCP connections for first round and R-multicast for the last round of protocol. Combining failure detector with ISIS was tricky and we have tried to explain our approach in section 3. There is just one parameter to ISIS algorithm i.e. the timeout for reply of first round message. Currently set to 5000 ms using the same reasoning as described for timeout value of failure detector.

3 Total ordering with failure detector.

In ISIS algorithm, the agreed messages to deliver are taken from the head of hold back priority queues of each process. In the three rounds multicast procedure, if a process failed before it can reliably broadcast the agreed priority to every other process, the leftover message will stay in the priority queue forever and block its following messages. Therefore, upon receiving a message from failure detector, all messages remained in priority queue from the failed process shall be removed. Note that it is possible that due to network delay, the failed process actually broadcast the agreed on priority but it comes after the failure message. Therefore, the deletion of "dead message" in the priority queue is actually performed after a reasonable amount of time period after receiving failure message. The heuristic time period we adopt right now is multiple times of maximum network delay(100ms) in this mp. The reasoning behind the selection of time out period is that, in the extreme case the sender process might failed during reliable multicast process, and only sent to one process. In this case, the process who received the priority will propagate this message to other processes but it requires at least two single trip time for the final priority to arrive. Therefore, the time out period is set to be 5-6 times of maximum network delay.

4 Implementation

Basic architecture and data flow at one process is shown in figure 1. As seen in the figure, an abstraction called Messenger is used to communicate with rest of the processes. This abstraction was helpful because it allowed rest of the code to think in terms of sending messages to processes instead of having to deal with low-level details of opening connections, serializing messages etc. This also allows us to switch network protocols easily without changing most of the code. Messenger is used by basic multicast to send the message to everyone, by ISIS to ask for proposed priorities in the first round and by failure detector to ping other processes.

Reliable multicast uses basic multicast as building block and ISIS uses reliable multicast to multicast agreed priority to everyone. One issue with reliable multicast is that we need a way to uniquely identify a message across the network. We use UUID class of Java standard library to create universally unique identifiers for the messages.

Failure messages flow from failure detector to ISIS to application. There is no direct communication between failure detector and chat application.

We have tried to keep our code modular and build up from small pieces: you will see a lot of interfaces and classes. We wanted to include a class diagram in the report but the report is already bigger than recommended size and the focus of the MP is to evaluate us more on the choice and design of the protocols rather than software engineering side of it. Nevertheless, we have provided proper documentation for all the classes, interfaces, methods in the code. We think that it shouldn't be tough for someone to dive in code and understand various components with the help of documentation.

5 Empirical evaluation of failure detector bandwidth

In figure 2 we have plotted averaged peak bandwidth(peak at each process averaged over all processes) for failure detector vs number of processes in the system. For fixed k=2 ping targets, once we reach the stage where total number of processes are equal to k, bandwidth usage is more or less constant. This validates the theoretical claims made by us in section 2.2.1 that SWIM uses constant bandwidth.

6 Acknowledgement

We would like to thank the entire staff for staying on top of our Piazza (public and private) posts and always providing useful feedback and hints. We would also like to thank Professor Indranil Gupta for the idea of SWIM failure detector.

7 Conclusion.

It was great experience implementing some of the algorithms studied in class and discovering some on our own. It helped us appreciate the fact that distributed protocols look easy on paper but have a lot of intrinsic details which come out only while implementing. We feel that in real world instead of directly implementing distributed protocols, we should use/have some kind of framework which among other things should make the testing in a distributed environment easy. We are not happy about the fact that most of the testing was adhoc and not in a standard way, this of course makes our application more bug prone. It will be great to have a lecture in class on proper ways to test and debug distributed applications.

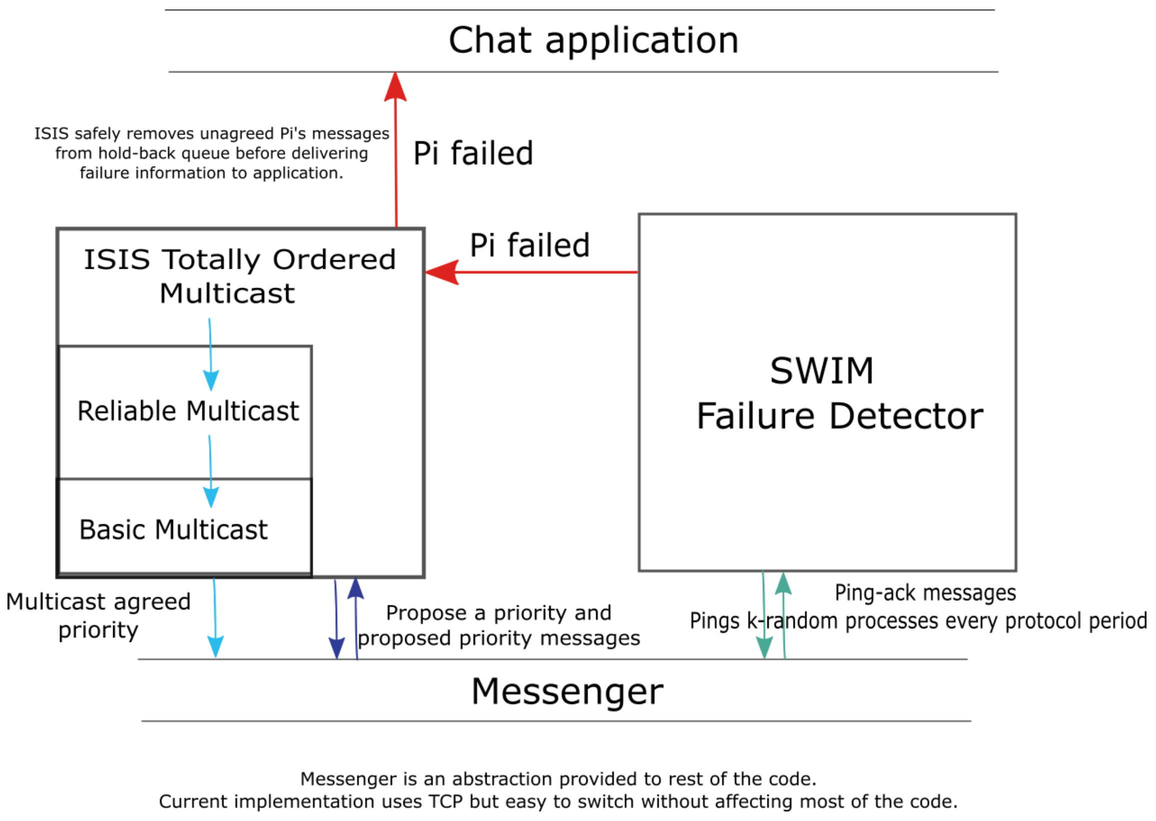
Chat application
ISIS safely removes unagreed Pi's messages from hold-back queue before delivering failure information to application.
Pi failed
Pi failed
ISIS Totally Ordered Multicast
Reliable Multicast
Basic Multicast
SWIM Failure Detector
Multicast agreed priority
Propose a priority and proposed priority messages
Ping-ack messages
Pings k-random processes every protocol period
Messenger
Messenger is an abstraction provided to rest of the code.
Current implementation uses TCP but easy to switch without affecting most of the code.

Figure 1: Architecture

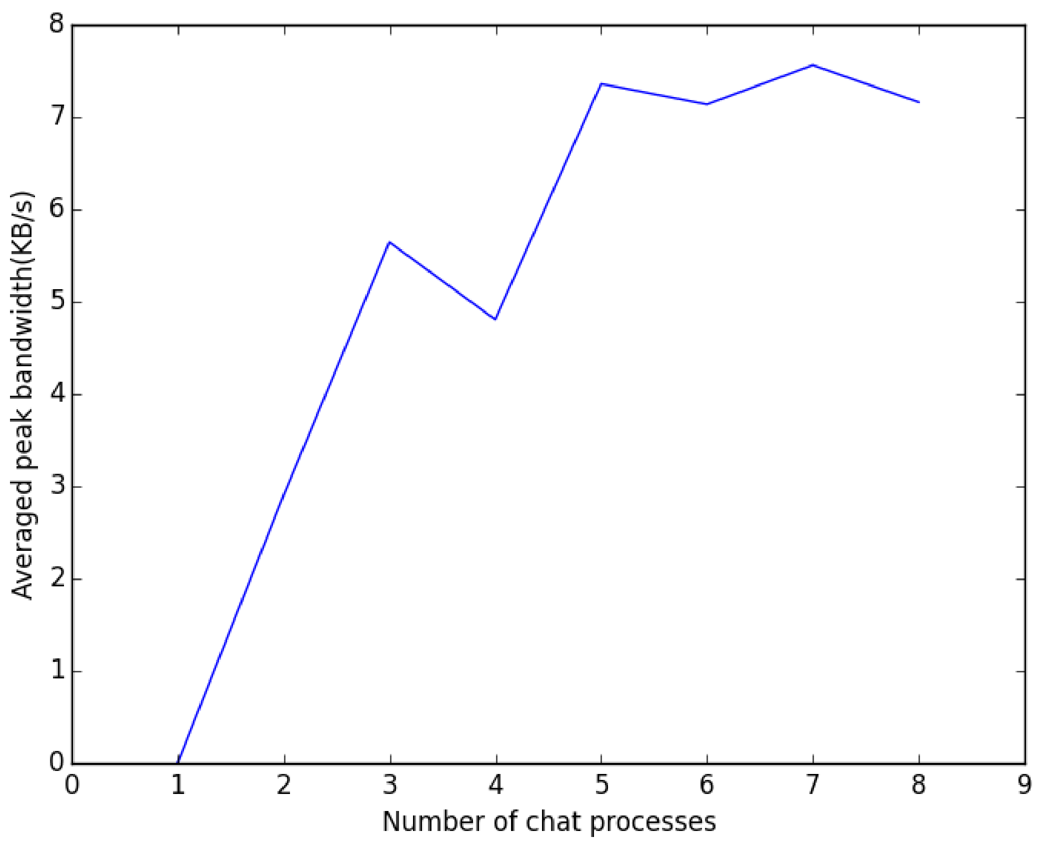

Figure 2: Bandwidth against number of process