Group Communication (Atomic Multicast)

Nasrin Seifi
230137517

Computer Science Department
University of Northern British Columbia

Winter 2020

Introduction:

Many modern on-line applications require scalable performance and high availability. Designing systems that combine scalability and fault tolerance, however, is challenging. Some systems respond to the challenge by weakening the guarantees they offer to the clients. While weak consistency has proved successful in some contexts, it is not appropriate to every application and often places the burden on the clients, who must cope with non-intuitive application behavior. Strong consistency leads to more intuitive application behavior but requires client requests to be ordered across the system before they are executed by the servers. [1]

Atomic multicast (AMCAST from now on)is a communication building block that allows messages to be propagated to groups of processes with reliability and order guarantees. It is one of the most useful primitives used in the design and the implementation of fault-tolerant systems. In AMCAST, all the messages multicast are delivered in the same order in every process. AMCAST ensures that the correct processes agree either to deliver or not the message and no two correct processes deliver any two messages in a different order. Because messages can be multicast to different sets of destinations and interleave in non-obvious ways, implementing message order in a distributed setting is challenging. Some AMCAST protocols address this challenge by ordering all messages using a fixed group of processes or involving all groups, regardless of the destination of the messages. To be efficient, an AMCAST algorithm must be genuine: only the message sender and destination processes should communicate to propagate and order a multicast message. A genuine AMCAST is the foundation of scalable systems, since it does not involve all processes.[2][3]

Problem Statment:

We consider an asynchronous message-passing system consisting of a finite set of N processes P. A process is correct if it never crashes, and faulty otherwise. Processes are connected by reliable FIFO channels, i.e., messages are delivered in the FIFO order, and every message sent by a process P to another process Q is guaranteed to be eventually delivered by Q provided both P and Q are correct. We fix $G \in P$ to be a set of process groups and let $|G| = k$. We assume that the process groups are disjoint, i.e., $\forall g_1 , g_2 \in G.\ g_1 \cap g_2 = \emptyset$. The assumption of disjoint groups is standard for practical multicast protocols. We consider the problem of implementing AMCAST in the above system, which allows a process to send an application message m from a set M to a set of destination groups $dest(m) \subseteq G$. We denote the events of multicasting a message m and delivering it by multicast(m) and deliver(m), respectively. For simplicity, we assume that all messages multicast in a single execution are unique. A message m is concurrent with a message m_0 if m_0 is multicast before m is partially delivered, and m is multicast before m_0 is partially delivered. Two messages m and m_0 are conflicting if $dest(m) \cap dest(m_0) \neq \emptyset$. An algorithm is a correct implementation of AMCAST if its every run satisfies the following: Validity, Integrity and Ordering. [6]

Solution:

To address the AMCAST issues, we are primarily concerned with halting failures, whereby a process stops executing without performing any incorrect actions. The term failure denotes a halting failure: A process ceases execution without taking any (visible) incorrect or malicious actions. In fault-tolerant systems, it is frequently necessary for the members of a group of processes to be able to monitor one another. They can then take actions based on failures, recoveries, or changes in the status of group members. [4]

The protocol we are considering to implement the AMCAST is based on a two-phase protocol by D. Skeen (unpublished communication, Feb. 1985). The protocol maintains a set of priority queues for each process, one for each AMCAST label, in which it buffers messages before placing them on the delivery queue. We assume that priority values are integers, with a process ID appended as a suffix to disambiguate the priorities assigned by different processes. Each message in the buffers is tagged deliverable or undeliverable. We give its pseudocode in Figure1.

The protocol creates a total order on application messages by assigning them unique timestamps, computed similarly to Lamport clocks. Timestamps are pairs (t, g) of a non- negative integer t ∈ N and a group identifier g ∈ G. They are ordered lexicographically using an arbitrary total order on G, with a special timestamp ⊥ being the minimal timestamp. For a timestamp ts = (t, g) we let time(ts) = t. To multicast an application message m, a process sends it in a MULTICAST message to the destination groups dest(m) (line 6). Each process maintains an integer clock, used to generate timestamps. When a process in a group g_0 receives MULTICAST(m) (line 8), it increments the clock and computes a local timestamp of m at group g_0 as the pair of the resulting clock value and the group identifier g_0 . This timestamp can be viewed as g_0 's proposal of what the final timestamp of m should be; it is stored in a LocalTS array. The process keeps track of the status of application messages being multicast in an array Phase, whose entries initially store START. When the

```
1  clock ← 0 ∈ ℕ;
2  Phase[] ← (λk. START) ∈
              (ℳ → {START, PROPOSED, COMMITTED});
3  LocalTS[] ∈ ℳ → (ℕ × 𝒢);
4  GlobalTS[] ∈ ℳ → (ℕ × 𝒢);
5  Delivered ← (λk. FALSE) ∈ ℳ → {FALSE, TRUE}

6  multicast(m)
7      send MULTICAST(m) to dest(m);

8  when received MULTICAST(m)
9      clock ← clock + 1;
10     LocalTS[m] ← (clock, g0);
11     Phase[m] ← PROPOSED;
12     send PROPOSE(m, g0, LocalTS[m]) to dest(m);

13 when received PROPOSE(m, g, Lts(g))
     for every g ∈ dest(m)
14     GlobalTS[m] ← max{Lts(g) | g ∈ dest(m)};
15     clock ← max{clock, time(GlobalTS[m])};
16     Phase[m] ← COMMITTED;
17     forall {m' | Phase[m'] = COMMITTED ∧
                    Delivered[m'] = FALSE ∧
                    ∀m''. Phase[m''] = PROPOSED ⟹
                          LocalTS[m''] > GlobalTS[m']}
         ordered by GlobalTS[m'] do
18         Delivered[m'] ← TRUE;
19         deliver(m');
```

Fig. 1. Skeen's protocol at a process $p_i \in g_0$.

process computes a local timestamp for m, it advances m's phase to PROPOSED . It then sends the local timestamp in a PROPOSE message to all the destinations of m (including itself, for uniformity). A process that is a destination of m acts once it receives a PROPOSE message for m from each destination group g ∈ dest(m), which carries m's local timestamp Lts(g) at g (line 13). The process computes the final global timestamp of m as the maximal of its local timestamps and stores it in a GlobalTS array. The process also advances the phase of m to COMMITTED and ensures that its clock is no lower than the first part of the global timestamp. Note that all destinations of m will receive the same sets of local timestamps for m and will thus compute the same global timestamp. Additionally, global timestamps are unique for each application message: if two messages got the same global timestamp (n, g), then they must have got the same local timestamp from group g; but this is impossible because a process increments its clock when issuing a local timestamp (line 9). Having computed the global timestamp for m, the process tries to deliver one or more committed messages (line 17). A Boolean array Delivered keeps track of whether a given message has been delivered. Messages are delivered in the order of their global timestamps; hence, the process can deliver a message m' only if it has already delivered all messages addressed to it with a lower global timestamp. A subtlety is that the process does not know the global timestamps for the messages m" that are in the PROPOSED phase. Hence, the process only delivers m' if all such messages m" have local timestamps higher than the global timestamp of m' : then their global timestamps will also be higher than that of m'. Note that this check is complete: application messages the process will receive for multicasting after delivering m' will get global timestamps higher than GlobalTS[m']. This is because, when the process commits m' , it advances its clock so that it is no lower than GlobalTS[m'] (line 15). Thus, any application message the process receives afterwards will get a local timestamp at g0 higher than GlobalTS[m'] and, thus, will also get a global timestamp higher than GlobalTS[m'].
Theorem 1: Skeen's protocol in Figure 1 is a genuine implementation of atomic multicast among singleton groups. Note that in Skeen's protocol a process can increase its clock at any time without violating correctness. We use this insight to construct a fast fault-tolerant version of this protocol.[4][5][6]

Software Architecture:

Based on the general architecture, we have some basic function like Create Group, Add Member, Create Messeage, Multicast, Receive, Proposed, Deliver, etc.
We make some groups in which each group has a fixed number of processes |G|=k, each process interact with each other in the group. The first method we are using is for defining each group and its members. We expect to display the groups and their members and the members' attributes like status, local timestamps, queues, etc. The GUI interacts with all functions to manage the visualization part. The next step will be, choose a process, call the Create(message) function, then Multicast(message) and then all functions in the algorithm will be used in the

same order in Figure1. After changing the status of the message (from the first send till pop from delivery queues) in each step we expect the GUI displays the updated members' attributes. So, GUI will interact with all functions from the first step. The Table below show the relationship between functions and their orders.

First Step	2nd Step	3rd Step	4th Step	5th Step	....	Final Result
Create Group → Add Member	Choose a Group	Choose a Process	Create (m)	Multicast (m)	(The Protocol role) Display the updating status	Show the message received by all members

Implementation:

To implement the AMCAST, we use JAVA language programming and for the visualization part, we use JAVAX library which we have used this language programming before. To simulate it, there are two options, the first one is simple we can pick a process and then click on the "multicast" to see the result or we provide the scenarios that cover the problems and then click on the "scenario" to run, in the scenario part we can consider using socket programming as well to handle sending and receiving message, the second one is, in case of the time permits to go further, using one of the simulation applications like MiniNet or SimGrid and simulate the multicasting message in the virtual network and then use the log file as an input for our simulation to run the program.

The features we consider to add in case of implementing the basic AMCAST are: joining new members to a group while sending messages, crash a process during the casting message, and implement AMCAST with a Coordinator and without that.

Milestones:

- Implementation of Basic Algorithm for receiving and sending messages in the distributed system (23rd Feb.)
- Add missed Properties like only consider the members of the group for multicasting and handle the buffer queues (1st Mar.)
- Test to meet requirements for AMCAST(8th Mar.)
- Add GUI for visualization(22nd Mar.)
- Final Test (29th Mar.)
- Final Submission and provide report document (5th Apr.)

I make this plan to have more time in case of unexpected challenges happened and I missed one of the milestones, I would have more time to cover that in the coming week and present the project on time.

References:

1. Samuel Benz, Parisa Jalili Marandi, Fernando Pedone, Benoît Garbinato, "Building global and scalable systems with Atomic Multicast", Middleware ’14, December 08 - 12 2014, Bordeaux, France

2. Paulo R. Coelho, Nicolas Schiper, Fernando Pedone, "Fast Atomic Multicast", 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

3. Carole Delporte-Gallet, Hugues Fauconnier,"Fault-tolerant Genuine Atomic Multicast" , LIAFA, Universite Denis Diderot, 2, pl. Jussieu, F75251 Paris Cedex 05.

4. KENNETH P. BIRMAN and THOMAS A. JOSEPH, "Reliable Communication in the Presence of Failures", ACM Transactions on Computer Systems, Vol. 5, No. 1, February 1987, Pages 47-76.

5. Niklaus Hirt, "Atomic Broadcast on various middlewares", Diploma Thesis 1997/98

6. Alexey Gotsman, Anatole Lefort, Gregory Chockler, “White-Box Atomic Multicast”, arXive:1904.07171v1, 15 Apr 2019.