ASD Work Assignment 1

António Ferreira 58340, Dinis Silvestre 58763, Bravo Mota 61359
NOVA School of Sciences and Technology,
Lisbon

October 18, 2023

1 Introduction

In the present day, extensive systems' architectures must be distributed, whether for improved performance, failure tolerance, easy scalability, etc. However, there are challenges to the several components working as a single system. One of these problems is the propagation of messages/instructions that keep the different subcomponents in the same state. In this document, we report the design and implementation of a broadcast algorithm to propagate messages in a distributed system. We designed our implementation of a reliable broadcast algorithm based on practical classes and lectures of the ASD course. Our algorithm is based on gossip and has some optimizations to achieve anti-entropy, for lower load on the system, and HyParView to achieve efficient dynamic membership. TODO:[high level results + evaluation criteria]

2 Methodology & Solutions

2.1 The Reliable Broadcast

A Reliable Broadcast is one that ensures the delivery of all the messages to all correct processes. In theory, not worrying about performance, this would be easy to achieve with a Flood Broadcast Algorithm (Eager push), in which every process sends every message to every other process. The problem is that this puts a big load on the network, most of the communication in this model would be obsolete as each process would receive on average $\#\pi$ times the same message (π = nodes in the systems).

2.2 An improvement to Flood Broadcast (Epidemic broadcast)

A small improvement, would be to instead of transmitting messages to all π nodes, only do so for a fanout t, typically $t = ln(\#\pi)$. We implemented this algorithm ($Epidemic$) as a first stage of our project, strongly based on practical classes of the course. It eases the load on the network compared with the Flood Broadcast algorithm.

2.3 A better solution (Reliable Broadcast with Anti-Entropy)

An alternative that overcomes the problem of a big load in all processes and network is the Anti-Entropy model, which uses Pull Gossip. In this model, only messages requested are sent to processes, alleviating the load on the network. Each process periodically sends a request to a random peer, in which it sends it's list of delivered messages so the process receiving the request can send back the messages that it has but the requester doesn't. This algorithm is particularly a great improvement if messages in the system tend to be extensive. By only sending the IDs of the messages a process has, the actual content of messages only travels when confirmed to be necessary.

2.4 Membership protocol

The membership protocol we've employed is an implementation of HyParView, retaining its core concepts while making some minor adjustments for improved clarity, as explained in the pseudocode that follows. One notable departure from the original paper's pseudocode is in the concept of the activeView. In the original paper, it suggested adding nodes to this set immediately after establishing a unidirectional connection. In our implementation, we maintain separate sets for incoming and outgoing connections, only adding a node to the activeView when it appears in both sets.

When a node initially connects to the contact node, it sends a Join message, similar to the original pseudocode. Upon receiving the Join message, the node establishes a bidirectional connection and forwards Joins to nodes already in its active view. Another distinction is how communication is handled with the layer above. In our approach, each time a node is added to the activeView, a newNeighbor request is dispatched to the dissemination layer along with the new node. Subsequently, the majority of the protocol functions as outlined in the original HyParView paper.

2.5 Algorithms, pseudo-code

The pseudo-code for our algorithms, can be found in later pages of this document. Following, in general, the structure for pseudo-code though in classes.

3 Implementations

To implement we used the Babel framework on java wich simplified the conversion of our pseudo-code into actual implementations

4 Results

Based on our results, the Flood protocol exhibits the highest redundancy rates across all scenarios, indicating that it tends to provide the most redundancy in message propagation. We also conclude that faster rates of request lead to higher redundancy rates across all protocols, suggesting that faster communication speeds allow for more redundant message propagation. On the other hand, larger payload sizes generally result in reduced redundancy rates across all protocols, indicating that larger data payloads might slow down the message propagation process and reduce redundancy. We tested locally with 105 processes.

The Flood protocol appears to be the most effective in ensuring robust message transmission, even though it may result in increased network traffic. However, the Anti-Entropy protocol has an acceptable performance and may be better appropriate in cases when network resources are limited.

The redundancy rate is the number of repeated messages received by a process divided by the total number of messages received (repeated + distinct).

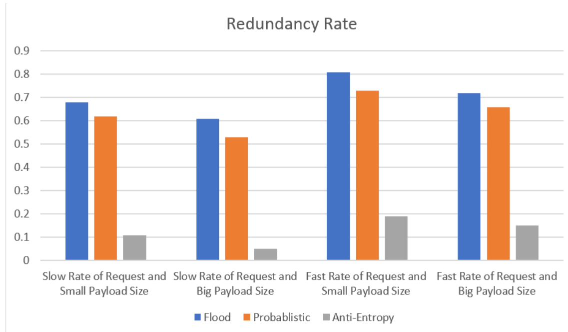

Figure 1: Redundancy rate for each broadcast considering four combinations of application parameters

5 Conclusion

We ensured that failure prevention conditions were met, such as when a neighbour fails, all three protocols correctly identify and respond to the failure by removing the failed neighbour from their neighbour lists. Furthermore, the Anti-Entropy protocol, in particular, demands missing messages in order to ensure message trustworthiness. To respond to channel failures, the protocols make use of the channel construction and notification methods. They ensure that messages are only sent when the channel is ready and that they recover from channel disruptions.

In summary, the Flood, Probabilistic, and Anti-Entropy protocols for broadcasting in a distributed network have been built and evaluated. The Flood protocol provides great message redundancy but may result in increased network traffic. The Probabilistic protocol prioritises redundancy over efficiency, whereas the Anti-Entropy protocol prioritises reliability over redundancy.

Some potential extensions include optimisation, security enhancements, assuring secure message propagation, and dynamic changes of parameters such as fanout and timeout thresholds based on network conditions.

References

[1] João Leitão, José Pereira, Luís Rodrigues (May 2007) Epidemic Broadcast Trees.

[2] Pedro Fouto, Pedro Ákos Costa, Nuno Preguiça, João Leitão (2022) Babel: A Framework for Developing Performant and Dependable Distributed Protocols

Algorithm 1 Epidemic Broadcast

```
Interface
    Requests :
        rBroadcastRequest(msg)

    Indications :
        rBroadcastDeliver(s, msg)

end Interface

State
    myself                                  ▷ My own address/port
    t                                       ▷ Fanout of the protocol, t = ln(n)
    neighbours                              ▷ known peers
    delivered                               ▷ Ids of messages already delivered
end State

// Event handlers
Upon Init (π, self) do
    myself ⟵ self
    t ⟵ ln(#π)
    neighbours ⟵ π
    delivered ⟵ {}
end Upon

Upon rBroadcastRequest (m) do
    Trigger rBroadcastDeliver ( s, m )
    mid ⟵ generateMid(m)
    delivered ⟵ delivered ∪ mid
    targets ⟵ randomSubsetWithSize(neighbours, t)
    for p ∈ targets do
        Trigger Send ( p, FloodMessage, mid, m, s )
    end for
end Upon

Upon Receive (FloodMessage, mid, m, s) do
    if mid ∉ delivered then
        Trigger rBroadcastDeliver ( s, m )
        delivered ⟵ delivered ∪ mid
        targets ⟵ randomSubsetWithSize(neighbours, t)
        for p ∈ targets do
            Trigger Send ( mid, m, p )
        end for
    end if
end Upon
```

Algorithm 2 Reliable Broadcast with Anti-Entropy

```
Interface
    Requests :
        rBroadcastRequest(msg)

    Indications :
        rBroadcastDeliver(sender, msg)

end Interface

State
    myself                ▷ My own address/port
    gossipTarget          ▷ target to send a Gossip Message to whom requested a message
    pi                    ▷ set of processes you send a message to
    delivered             ▷ Ids of messages already delivered
    deliveredMsgsMap      ▷ Map to keep track of delivered messages by neighbour
    msgContentMap         ▷ Map to keep track of the messages' content according by mid
end State

// Event handlers
Upon Init (self) do
    myself ⟵ self
    pi ⟵ {}
    delivered ⟵ {}
    deliveredMsgMap ⟵ {}
    msgContentMap ⟵ {}
    Setup Periodic Timer SendPullGossip
end Upon

Upon BroadcastRequest (m, mid, s) do
    msgContentMap ⟵ msgContentMap ∪ (mid, m)
    delivered ⟵ delivered ∪ mid
    Trigger rBroadcastDeliver ( s, m )
    deliveredMsgMap[s] ⟵ deliveredMsgMap[s] ∪ delivered
end Upon

Upon Timer SendPullGossip do
    if (pi ≠ {}) then
        p ⟵ randomSelection(pi)
        Trigger send (p, PullGossipMessage, delivered)
    end if
end Upon

Upon Receive (PullGossipMessage, requesterDelivered, s) do
    missingMessages ⟵ delivered \ requesterDelivered
    for mid ∈ missingMessages do
        Trigger Send ( s, GossipMessage, mid, msgContentMap[mid])
    end for
end Upon
```

Upon Receive ($GossipMessage$**, mid, m, s) do**
msgContentMap $\longleftarrow$ msgContentMap $\cup$ (mid, m)
delivered $\longleftarrow$ delivered $\cup$ mid
Trigger rBroadcastDeliver (s, m)
deliveredMsgMap[s] $\longleftarrow$ deliveredMsgMap[s] $\cup$ delivered
end Upon

Upon NeighbourUp (p) do
pi $\longleftarrow$ pi $\cup$ p
end Upon

Upon NeighbourDown(p) do
pi $\longleftarrow$ pi $\setminus$ p
deliveredMsgMap $\longleftarrow$ pi $\setminus$ p
end Upon

Algorithm 3 HyParView

```
Interface
    Indications :
        NeighbourUp(p)
        NeighbourDown(p)
        ChannelCreated(id)

end Interface

State
    outgoingConnections                ▷ set with all outgoing connection
    ingoingConnections                 ▷ set with all incoming connection
    activeView                         ▷ set with all established bidirectional connections
    passiveView                        ▷ set will serve as "backup" for the activeView
    ARWL                               ▷ active random walk length
    PRWL                               ▷ passive random walk length
end State

// Event handlers
Upon Init (contactNode) do
    openConnection(contactNode)
    outgoingConnection ⟵ {}
    ingoingConnection ⟵ {}
    activeView ⟵ {}
    passiveView ⟵ {}
end Upon

Upon outConnectionUp (node) do
    openConnection(contactNode)
    if node ∈ ingoingConnections then
        activeView ⟵ activeView ∪ node
    end if
    if #outgoingConnections = 1 ⋀ # outgoingConnections = 0 then
        Trigger Send (node, JoinMessage, self)
    end if
end Upon

Upon outConnectionDown (node) do
    outgoingConnections ⟵ outgoingConnections \ node
    if node ∈ activeView then
        activeView ⟵ activeView \ node
        passiveView ⟵ passiveView ∪ node
    end if
end Upon

Upon inConnectionDown (node) do
    ingoingConnection ⟵ ingoingConnection \ node
    activeView ⟵ activeView \ node
end Upon
```

```
Upon inConnectionUp (node) do
    ingoingConnection ⟵ ingoingConnection ∪ node
    if node ∈ outgoingConnection then
        activeView ⟵ activeView ∪ node
        Trigger NeighbourUp ( node )
    end if
    if #ingoingConnection > 1 then
        Call establishOutgoingConnection(node)
    end if
end Upon
Upon Receive (Join, newNode) do
    Call establishOutgoingConnection(newNode)
    for n ∈ activeView ⋀ n ≠ newNode do
        Trigger Send ( ForwardJoin, n, newNode, ARWL, myself )
    end for
end Upon
Upon Receive (ForwardJoin, newNode, timeToLive, sender) do
    if timeToLive = 0 || #activeView = 1 then
        Call establishOutgoingConnection(newNode)
    else
        if timeToLive = PRWL then
            Call addNodePassiveView(newNode)
        end if
        n ⟵ selectRandom(activeView\sender)
        Trigger Send ( ForwardJoin, n, newNode, timeToLive-1, myself )
    end if
end Upon
Upon Receive (Disconnect, node) do
    if node ∈ activeView then
        activities ⟵ activeView \ node
        Call addNodePassiveView(node)
    end if
end Upon

Procedure establishOutgoingConnection (node)
    if self ≠ node ⋀ node ∉ activeView then
        if #activeView = activeViewCapacity then
            Call dropRandomElementFromActiveView
        end if
        openConnection(node)
    end if
end Procedure
Procedure dropRandomElementFromActiveView
    n ⟵ selectRandom(activeView)
    Trigger Send ( Disconnect, n, myself )
    closeConnection(n)
end Procedure
Procedure addNodePassiveView(node)
    if node ≠ myself ⋀ node ∉ activeView ⋀ node ∉ passiveView then
        if #passiveView = passiveViewCapacity then
            n ⟵ selectRandom(passiveView)
            passiveView ⟵ passiveView \ n
        end if
        passiveView ⟵ passiveView ∪ node
    end if
end Procedure
```