*Montrèal August 9, 2016*



Concordia University

Department of Computer Science and Software Engineering

COMP 6231 – Distributed System Design

Project Design Document

Highly Available Distributed Staff Management System

From:

*Klajdi Karanxha – 26173780*

*Karl Samaha – 21973495*

*Richard Smith - 27859279*

*Keyvan Kamali – 40016025*

*Mohit Rana - 40017462*

To:

Professor Sukhjinder K.Narula

**TABLE OF CONTENTS**

Introduction 2

Architecture Description 3

Relevant Design Points 4

Total order broadcasting 4

Group Leader Process 6

Reliable FIFO Broadcast 7

Failure Detection Subsystem 8

Leader Election Subsystem 10

Scenario Description and Testing 12

Client Side 12

Front-End implementation and behaviour 12

Server Side 13

Closing Notes 14

# Introduction

The purpose of this project is to create a staff management system leveraging the potential of CORBA and UDP. The solution comprises many distributed entities, namely, the client side, where clients communicate to a front-end through CORBA. The goal is to implement a highly available system, which tolerates process crashes using unreliable failure detection. It is assumed that system tolerance for software failures is out of scope. In our design, there is a group of three replicas of a server process that run on different hosts in a redundant mode, offering high availability by periodically checking each other for failure detection.

A process in the group is the elected leader and receives requests from clients through a CORBA front end and sends responses back to them through the same front end. The leader of the group broadcasts the client request to all the servers in the group using a reliable FIFO broadcast mechanism atomically, and receives the responses from them. Once this ‘synchronization’ is done, it then sends a single response back to the client as early as possible. Since the **replicated servers are on a local area network**, they communicate using the unreliable UDP protocol. The communication between processes is made reliable by using FIFO reliable broadcast on top of the unreliable UDP protocol.

# Architecture Description

Clients communicate to a Front-End entity using CORBA. The Front-End entity, in turn communicates to a process leader (the leader) using the UDP protocol. The client sends requests to the Front-End, which in turn communicates and sends requests to the process leader.

The leader does two main functions with the request: ***a)*** it broadcasts it using UDP to all the other replicas and ***b)*** it will execute the transaction itself on its own (the replicas will do the same).

The communication beyond client – front-end (where client to front-end is CORBA) is through the UDP protocol. The UDP protocol is unreliable; hence we need to enhance the middleware’s reliability while sending and acknowledging messages between processes. Below is a description of the main components of our system design and the solutions architecture:

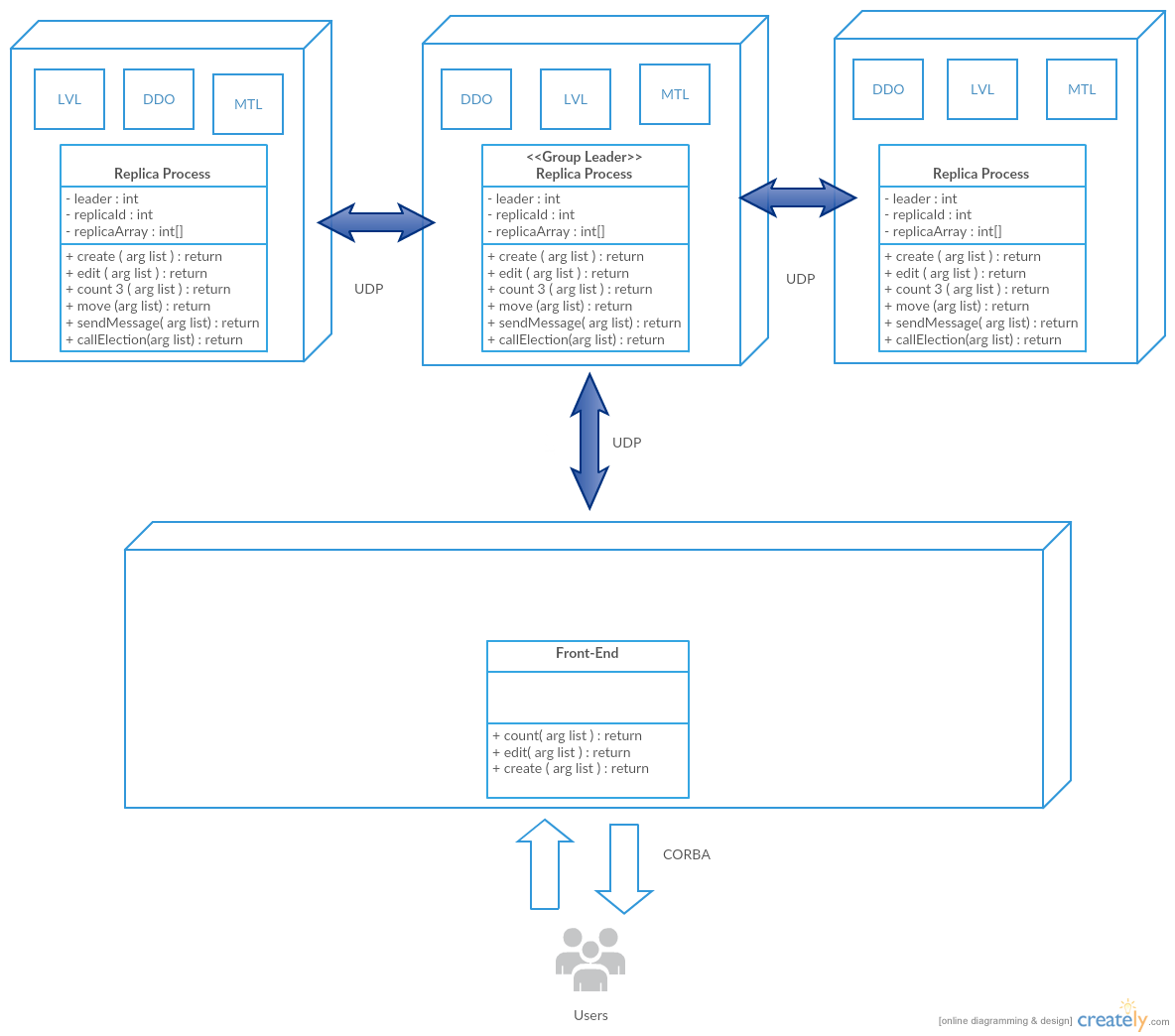


Figure Overall Architecture of the System

## Relevant Design Points

### Total order broadcasting

What is the total order broadcast and whether that is covered in their system.

The goal of our system is to broadcast requests reliably over the replicas, for that purpose we have implemented total order broadcasting. When the sender atomically broadcasts messages among the replicas that mean that the messages are in a total order sequence where endpoint processes first message sent by the sender before the second message sent by the same sender. In addition it is guaranteed that all the participants have processed a message before the sender dispatches the next one. By those two points we ensure that we have a valid system, which has achieved integrity. By integrity we mean that each message is delivered to each recipient at most once after it has been broadcast. Currently in our system, we multicast a certain packet in the sequence and piggyback the replies until we get the number of acknowledgements from all the parties. Meanwhile, we just drop the redundant replies.

To cover total order in an actively replicated system like ours (with the high availability goal in scope) we enforce strict sequential of the execution of client requests which makes the requests transactional.

FIFO broadcast ensures that the packages are delivered to the endpoint reliably, which covers the total ordering as it respects the sequence and the order of the processing.

Our system tolerates **n-1** crashes for **n** replicas; it means that even if there is one replica left, namely the leader, our system will function as one.

Our replicas have a separate thread to emit Pings and another separate thread to listen to pings. Our replicas can server process multiple requests at the same time and will eventually queue them up. However, we have one process Replica and we do not multithread this process.

## Group Leader Process

**Responsibility: Keyvan Kamali, Mohit Rana**

The responsibility of this component is to provide the implementation of the group leader process, which receives a request from the Front-End and will implement the FIFO broadcasts, which will reliably transmit all requests to all the server replicas in the group using UDP. This is done through the use of a UDP listener thread, which only gets started in the group leader process. Once a new leader gets elected, this leader also sends a message to the Front End to notify it of the newly elected leader.

The client side represents mainly managers who want access the clinic server for transactional purposes and oblivious of the replication and redundancy, they only interact with the front-end through CORBA. The process group leader interacts between front-end and server replicas side.

The responsibilities of group leader involve receiving request from clients and sending them to the replicas by FIFO broadcasting them. The group leader interacts with a FIFO broadcast component which will queue its requests and broadcast them reliably though UDP. The process leader must have interface to a FIFO implementation which can represent a queue that pushes the requests on its tail order-wisely. The process leader initializes a listener which checks the queue periodically and broadcast received requests to all the replicas until the queue is emptied or another special condition is met.

UDP datagrams are compared and if all servers return the same response to the leader, it sends a single acknowledgement to the front end. The leader process also, as all the other processes, consistently checks the vitals of the other servers and broadcasts the fact that one response may be missing, transitioning to the election subsystem only when a leader is down or unresponsive.

## Reliable FIFO Broadcast

**Responsibility: Richard Smith**

The FIFO broadcast mechanism over UDP multicasting is interfaced by the process leader and will forward the request to other processes in the group whenever it receives them from the Front-End.

The FIFO broadcast subsystem interfaced by the group leader process can handle concurrent requests. Reliability of UDP shall be reached by ordering and sequencing the concurrent requests received by the FIFO subsystem. The order will be priority-based where first requests have higher priority than later requests and they are ordered in the way they are received. That means that our FIFO subsystem will turn the concurrent requests into sequenced stream of requests, thus achieving the reliability of transmission over UDP. The order on which the messages are delivered is not guaranteed and is based on first-come-first-serve basis.

Lastly, the FIFO broadcast will have additional acknowledgement requests used for ‘handshaking’, in order to ensure that no answer is returned unless all the processes are ready to process the next packet in sequence. We will use a hold back and delivery queue in order to queue up the packets that are received by some replicas but not the others and deliver them according to their ordering number.

The process leader will broadcast the message to the interfaced FIFO subsystems on the other replicas in the group.

The receiver processes, interfacing to FIFO will behave the same by ordering the packets; processing the message and send the response to the leader process, which will then transmit it, back to the Front-End (since the Front-End is a single running instance, the message can be unicasted in this case).

## Failure Detection Subsystem

**Responsibility: Klajdi Karanxha**

The failure detection subsystem is responsible for identifying processes that do not reply to a ping monitor routine and flag those processes (and their ID’s) as failed. For the sake of this project we assume that there will be at least one server replica up and running and that system behaviour produced from all the processes failing at the same time shall be out of this project’s scope. Failure detection subsystem is part of each replica independently so that each process is responsible for inspecting (pinging) all the rest of the other processes on a regular routine timeout (e.g. every *x* ms). If the ping receives no response within a pre-set interval of time then the ping originator (the process who launched the ping) will flag the unresponsive process as failed. The failed servers reported, will be multicast in order to achieve ‘consensus’ on a failure.

For the scope of this project we assume that once a process has failed the attempt to bring it up is abandoned, hence no pinging is any longer necessary once a process is flagged as failed. If the failed process is the leader process, the system flow transitions to the leader election subsystem automatically, by triggering as such a new ‘leader election’.

We are using **Ping-Ack** protocol to detect failures. Basically Process *Pi* is querying process *Pj* every γ units and if *Pj* doesn’t reply within and additional γ units then *Pi* will mark Pj as a suspect up to the point where all the Pj can be a suspect. If all other processes are suspected then *Pj* will be marked as failed on timeout, else *Pi* will request consensus from other peers to mark *Pj* as failed.

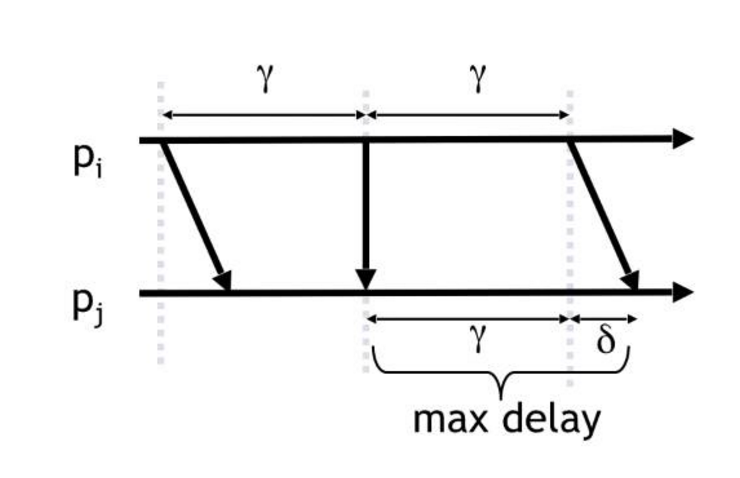


Figure 2 - 'Gamma' period after timeout, every process will notice absence of response

In our implementation, after the timeout, eventually every process will notice the absence of a reply ***PING*** packet and suspect the failed process within a time interval δwhich is less than the timeout γ.

We achieve **completeness** by basically keeping every crashed process in the suspect list. In our case it is **strong completeness** because all the correct processes eventually suspect every crashed process. We also achieve **strong accuracy** by ensuring that process that come back online within the timeout are removed from the suspects list, and eventually all live processes are never suspected. In the end the election process takes advantage of the failure behaviour to transition to the election of a new leader.

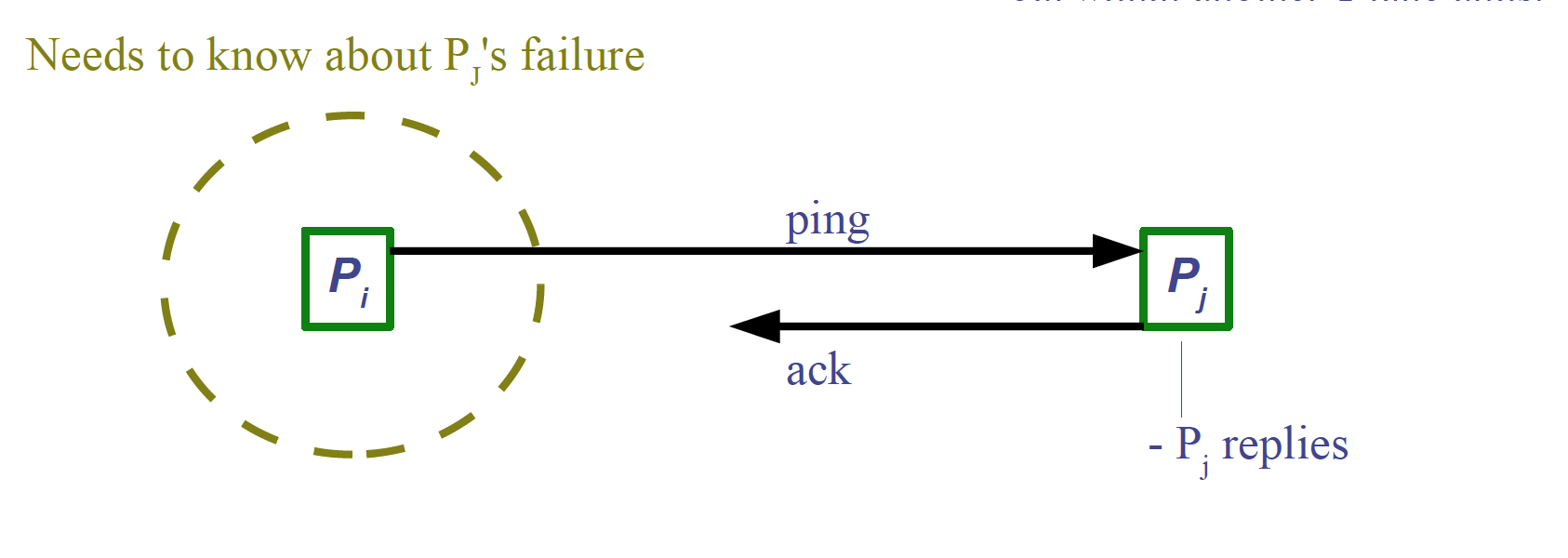


Figure 3 - Ping-Ack protocol

We implemented an **eventually perfect failure detector.** Where we have strong completeness and eventual strong accuracy. We basically implement two events, suspect/unsuspected as an ‘entry’ point to the failure detection logic. Each replica has two threaded properties, listener and emitter. One of them (listener) listens for ‘ping’ packets and replies back and the other emits ‘ping’ packets or ‘ping+’ when it is asking consensus to mark a suspected process as failed. The failure-detection algorithm we use is peer-to-peer and each replica receives pings at a special (*6000+ replica\_id*) port.

## Leader Election Subsystem

**Responsibility: Karl Samaha**

The responsibility of this subsystem is to elect a leader process in the event the current one fails. During the ‘election’ the client is waiting for a response while the new leader is chosen and then returns the response to the Front-End, as well as the remaining replicas as expected with additional information for logging purposes (that mention that a new leader is now elected) and because front end must know at all times the leader process.

We will eventually use “The bully algorithm” as the strategy for leader election where the new group leader will be selected based in a pre-determined criteria (e.g., since they are all redundant, higher process ID is the constraint for choosing a new leader). Obviously, the election subsystem will announce the leader to the Front-End. The bully algorithm allows processes to crash during an election, although it assumes that message delivery between processes is reliable. The algorithm also assumes that each process knows the identifier of others, and that it can communicate with all such processes for failure detection purposes.

There are three types of messages in this algorithm: an election message is sent to announce an election, an answer message is sent in response to an election message and a coordinator message is sent to announce the identity of the elected process – the new coordinator in our case. A process begins an election when it notices, through the failure detection algorithm, that the coordinator has failed. Several processes may discover this concurrently. Hence in every replica process­, a list with all replica processes, each associated with it’s own identifier will be available, along with a variable “leader” to determine the group coordinator.

The process that knows it has the highest identifier can elect itself as the coordinator simply by sending a coordinator message to all processes with lower identifiers. If a process with a lower identifier notices a failure, it can begin an election by sending an election message to those processes that have a higher identifier and awaiting a time T for an answer message in response. If none arrive, it is assumed that all higher-ordered processes have failed, considers itself the coordinator and sends a coordinator message to all processes with lower identifiers announcing this. Otherwise, if the process does receive answer messages back, it waits a further period T’ for a coordinator message to arrive from the new coordinator. If none arrive, it begins another election. The condition for a bully algorithm to run is that the leader process has failed and is unresponsive.

If a process pi receives a coordinator message, it sets its variable elected­ to the identifier of the coordinator contained within it and treats that process as the coordinator. If a process receives an election message, it sends back an answer message and begins another election, unless it has begun one already. When a process is restarted after a failure, it must begin an election. If it has the highest process identifier, then it will decide that it is the coordinator and announce this to the other processes, hence “bullying” out the current coordinator.

# Scenario Description and Testing

While we are still working on the definition of a robust unit testing strategy and test cases we have identified the components to test and the end-results that we wish to achieve.

The scenarios, which we would like the system to accomplish, are to be described as following:

## Client Side

### Front-End implementation and behaviour

#### Logging and execution of the Front-End instance.

* Scenario: Client attempts to connect to front end
  + Given font end is active
  + When client attempts connects to front end
  + Client address, submitted credentials, timestamp, and authorization success status are logged
* Scenario: Request from clients and responses from replicas are sent and logged
  + Given client is connected to front end
  + When client issues a request
  + Then request from client, client address, manager id, timestamp, and success status is logged by front end
    - And request sent to current group leader is logged with leader ID/address
    - And response status/timeout and timestamp from group leader is received and logged
    - And response sent to client, client address, manger id, timestamp, and success status is logged
* Scenario: Leader is assigned/reassigned
  + Given the leader replica is unassigned/invalid
  + When replicas conduct an election
  + Then the result is reported by the new leader to the front end
    - And previous leader is logged
    - And new leader response, address, and id is logged
    - And new leader is assigned
* Scenario: All replicas are registered with root name service / orbd
  + Given the front end is registered with a root name service
  + When a replica attempts to contact the front end
  + Then it must be registered with the same root name service /orbd

## Server Side

#### Normal request handling and logging in a multithreaded environment

* Scenario: Creation of Record
  + Given replica is connected
  + When request is received from leader
  + Then request is logged
    - And record is created
    - And result is logged
    - And result is returned to leader
* Scenario: Edit of a Record
  + Given replica is connected
  + When request is received from leader
  + Then request is logged
    - And record is created
    - And result is logged
    - And result is returned to leader
* Scenario: Count of Records
  + Given replica is connected
  + When request is received from leader
  + Then request is logged
    - And record is created
    - And result is logged
    - And result is returned to leader
* Scenario: Transfer of Record
  + Given replica is connected
  + When request is received from leader
  + Then request is logged
    - And record is created
    - And result is logged
    - And result is returned to leader
* Scenario: Reliable Broadcast
  + Given a request from leader
  + When an acknowledgement sent
  + Then request is not performed until leader authorizes commit
    - And result is logged and returned to leader
* Scenario:
  + Bully Algorithm
    - Given a leader process failure
    - Trigger election
    - Force the selection of next higher id process
    - Test the establishing of a new leader
* Scenario:
  + Test the FIFO broadcast
    - Ensure that many packets are received at once.
    - Test that FIFO queue is queuing them and broadcasting them reliably.
    - Test the hold-up queue.

#### Replica failure detection

* Scenario: Ping monitor routine
  + Given replica is active
  + When timeframe is elapsed
  + Then ping is sent to other replicas
    - And request and response are logged
* Scenario: Ping listener routine
  + Given replica is active
  + When ping is received
  + Then response is sent
    - And request and response are logged
* Process Failure flagging
* Process Failure
  + A normal process fails (it is announced and flagged)
  + A process leader fails (it is announced and flagged) triggering an election.
    - Group Leader Election (bully algorithm)
    - New leader notification to Front-End

# Closing Notes

We are planning to use git to keep source code revisions.