18-749 Building Reliable Distributed Systems

Electrical and Computer Engineering Department
Carnegie Mellon University
priya@cs.cmu.edu

The Project Guide

OVERVIEW

Objective

The objective of the project is to become familiar with building reliable distributed systems, and to obtain hands-on experience in developing as many components of a real-world Distributed Fault-Tolerance standard as possible. This includes concepts such as replica consistency, logging, checkpointing, active replication, and passive replication.

Approach

The project will be done in groups of 5 members. Each group will have a private Slack channel for all day-to-day communication within the group, and between the group and the course staff. The class-wide project discussions will take place on the **#project** Slack channel.

The lectures will provide the background material necessary to complete the project successfully. The project will involve the actual implementation (in Java or some other programming language) of a fault-tolerant distributed application through the use of replication. The project will run through the course of the entire semester, with the project being built up in phases, with each milestone adding a new fault-tolerance mechanism on top of the preceding milestone.

The project will culminate in a final demonstration of the Distributed Fault-Tolerance Standard for the application of your choice, where faults will be injected (replicas will be intentionally crashed) and recovered from (replicas will be intentionally re-launched) within a running distributed system. The expectation is that the system will continue to be resilient to faults and not experience any downtime during the demonstration.

> **There are multiple different ways to implement the concepts underlying the Distributed Fault-Tolerance Standard.** There is no such thing as a perfect implementation or a golden-standard implementation. What is suggested in this document is one possible implementation, along with one possible sequence of operations. This document seeks to provide some helpful suggestions and guidelines, and should not constrain your implementation or your creativity.

Project Scope

The scope of the project includes specific concepts taught in class:

- Crash faults and message-loss faults,
- Fault detection via heartbeats,
- Active replication with replica consistency,
- Passive replication with replica consistency,
- Checkpointing, logging, duplicate detection,
- Maintenance of group membership,
- Fault injection to observe the system operate under faults, and
- Re-introduction of failed replicas to observe how recovery works.

Project Milestones

The project consists of multiple implementation phases, with milestones at every phase, and an associated grading component with the milestone.

1. Simple client-server application with fault-detection and configurable heart-beats **(5% of grade; entire team gets the same grade)**
2. Actively-replicated server with single fault and single-replica failover **(5% of grade; entire team gets the same grade)**
3. Warm-passively replicated server with single fault, with primary failover to a backup **(5% of grade; entire team gets the same grade)**
4. Recovery for active replication and passive replication with checkpointing **(5% of grade; entire team gets the same grade)**
5. Entire distributed fault-tolerance standard, with injected faults and recovery **(5% of grade; entire team gets the same grade)**
6. Peer review from your team-mates **(5% of grade; each team member gets a different grade)**

> The overall project is worth 30% of your grade for the course: 25% comes from team effort, and 5% from your individual contribution to the project milestones. Your individual 5% in the project comes from peer review from your group members. Note that different group members will take turns being the project manager. There are 5 milestones. So, 5 project managers. The milestone's project manager will be responsible for ensuring that the milestone goes smoothly. That individual portion of the project grade will be assigned at the end of the semester.

SYSTEM ARCHITECTURE

> Students should be familiar with writing programs in Java/Python/C/C++, and should be familiar with network programming (sockets, RMI, etc.) in those languages or frameworks. The course project is programming-intensive and students are expected to be writing code and programming from week 2 of the course. The course itself does not teach programming or network-level programming, but focuses on distributed systems concepts, assuming that programming fundamentals have been acquired elsewhere. Students are free to choose a programming language to implement their entire project.

The Application

The application is a simple client-server application that consists of the following:

- A client-server application, with 3 independent clients and a single server that is replicated;
- 3 server replicas distributed across different machines that are connected via the same network, with each server replica located on a different physical machine;
- The clients are all located on a single machine;
- The clients are constantly communicating with the replicated server, in a continuous loop, so that the servers are constantly receiving and processing requests.

The Fault Model

We are focused on detecting, and recovering from, two types of fail-silent faults:

- Crash of a single server replica, and
- Loss of a message between a client and the server.

Fault-Tolerance Infrastructure

The fault-tolerance infrastructure is modeled after the Fault-Tolerance industrial standard that we studied in class:

- We are dealing with a distributed asynchronous system. So, there is no global clock.
- A Replication Manager (RM) that is responsible for the overall replication of the system, and is aware of all of the membership changes and is responsible for maintaining the resilience of the system;

- A Global Fault Detector (GFD) that is responsible for managing the Local Fault Detectors (LFDs), one on each machine in the distributed system, and for communicating any and all membership changes (the list of failed replicas and new replicas) to the RM;
- A Local Fault Detector (LFD), one on each machine in the system, that is responsible for heart-beating the replicas on its machine, and for communicating the crash of any its replicas to the GFD;
- There is periodic communication between the LFD and the GFD as the GFD periodically heart-beats all GFDs.
- There is periodic heart-beating from the LFD to its local server replica.
- There is communication from the GFD to the RM each time there is a membership change.

Fault-Tolerance Properties

There are properties we will want to change **dynamically** in the system, i.e., we will want to change these properties even as the system is running and without killing and restarting the entire system:

- The number of server replicas,
- The heart-beat frequency of each LFD to its server replica;
- The heart-beat frequency of the GFD to its (child) LFDs;
- The checkpointing frequency of the primary replica to its backup replicas

Assumptions

Given the duration of the semester, the project will inevitably make some simplifying assumptions:

- The server is deterministic, which implies that the server is not multi-threaded, does not have any local timers, random-number generators, etc.
- The Replication Manager is not replicated, and is an acceptable single point of failure.
- The Global Fault Detector is not replicated, and is an acceptable single point of failure.
- Each Local Fault Detector is not replicated, and is an acceptable single point of failure. In fact, the failure of an LFD is synonymous with the failure of the entire machine.
- There is no need for a Fault Notifier in the system.
- There is no need for local Factories on each machine in the system.
- We will implement total order via TCP/IP instead of using a UDP-based reliable multicast protocol.

HINTS FOR IMPLEMENTATION

Notation

- Server replicas have a unique *replica_id* with three identical replicas, S1, S2 and S3.
- Clients have a unique *client_id* with three independent clients, C1, C2, C3.
- Each server has a piece of state, *my_state*, that is updated by the requests that it receives from the clients, C1, C2, C3.
- Each server has an additional piece of state, *i_am_ready*, that indicates that it is ready to start accepting and processing client requests. The initial value of *i_am_ready = 0,* which means that the replica is not ready to receive requests. The value of *i_am_ready = 1* when the RM sets the value and when it knows the replica is ready and when it has the correct state. The replica is ready only in one of two states: (i) it is the first replica of its kind, in which case *i_am_ready = 1* from the beginning, and (ii) it has received the checkpoint from a fellow replica, in which case *i_am_ready = 0* in the beginning and *i_am_ready = 1* after the checkpoint has been received and processed.
- In passive replication, each server additionally has a variable called *checkpoint_num* that represents the number of checkpoints exchanged between the primary and backup replicas from the beginning of time.
- Make sure that every message has a timestamp prefixed to the message, e.g., when printing the message to the console, prefix the message with *[timestamp]* before printing the rest of the message.
- Make sure that every client message is uniquely identifiable in the system. Every client must maintain a *request_num* parameter as well as a *client_id* parameter, and these must be embedded in the payload of requests, and should also identify the *replica_id* of the server for which the request is destined.
- Make sure that every server response also embeds the *request_num* parameter as well as the *client_id* parameter in the payload of its replies, so that we can match requests to replies. Every message between the client and the servers is identified by the tuple: <*client_id, replica_id, request_num*> along with the direction of message flow *(request* OR *reply).*
- Make sure that every server replica prints out the value of *my_state* before processing an incoming request, and before returning a reply to the client.
- Make sure that the LFD maintains a *heartbeat_freq* parameter, as well as a *heartbeat_count* that it prints to its console window, essentially counting the number of heartbeats it has sent.

- The LFD's heartbeat messages, and the *replica_id* of every replica that it heartbeats, should be printed out on its console window, for every periodic heartbeat.
- Make sure that the primary replica and the backup replicas maintain a *checkpoint_freq* parameter, as well as a *checkpoint_count* variable, when you implement passive replication. The *checkpoint_count* variable counts the number of checkpoints that have been exchanged between the primary and the backup replicas. Every checkpoint between the primary and the backup replicas contains both the value of the primary's *my_state* variable as well as the primary's *checkpoint_count* variable.
- Make sure that the GFD maintains an array of members, *membership[]*, that lists all of the *replica_id*s of the current server replicas running in the system, along with a variable called *member_count* that represents the number of alive and healthy replicas.
- Make sure that the RM maintains an array of members, *membership[]*, that lists all of the *replica_id*s of the current server replicas running in the system, along with a variable called *member_count* that represents the number of alive and healthy replicas.
- Make sure that LFD sends the *replica_id* to the GFD every time a replica is added or removed.

Example of Notation in Action

Client C1 sends the identical request with *request_num* 101 to active replicas S1 and S2.

> **The console window of C1 should show the following messages:**
> *[timestamp] Sent <C1, S1, 101, request>*
> *[timestamp] Sent <C1, S2, 101, request>*
> *[timestamp] Received <C1, S1, 101, reply>*
> *[timestamp] Received <C1, S2, 101, reply>*
> *[timestamp] request_num 101: Discarded duplicate reply from S2*
>
> **The console window of S1 should show the following messages:**
> *[timestamp] Received <C1, S1, 101, request>*
> *[timestamp] my_state_S1 =* ??? before processing *<C1, S1, 101, request>*
> *[timestamp] my_state_S1 =* ??? after processing *<C1, S1, 101, request>*
> *[timestamp] Sending <C1, S1, 101, reply>*
>
> **The console window of S1 should show the following messages:**
> *[timestamp] Received <C1, S2, 101, request>*

> *[timestamp] my_state_S2 =* ??? before processing *<C1, S2, 101, request>*
> *[timestamp] my_state_S2 =* ??? after processing *<C1, S2, 101, request>*
> *[timestamp] Sending <C1, S2, 101, reply>*

Things to keep in mind:

- *<C1, S1, 101, request>* and *<C1, S2, 101, request>* are identical in payload.
- *<C1, S1, 101, reply>* and *<C1, S2, 101, reply>* are identical in payload, if S1 and S2 are deterministic. Thus, one of them can be discarded by C1 as a duplicate.
- *my_state_S1* and *my_state_S2* should be identical, **before** processing the request 101 from C1, if S1 and S2 are deterministic.
- *my_state_S1* and *my_state_S2* should be identical, **after** processing the request 101 from C1, if S1 and S2 are deterministic.
- C1 should automatically update its *request_num* to 102 (in preparation for sending the next request to the server replicas) after receiving the first reply to 101, basically, after receiving whichever one of *<C1, S1, 101, request>* and *<C1, S2, 101, request>* shows up first.
- *[timestamp]* is basically from the local machine that each process is running on. It's useless for virtual synchrony or replication, but it will be useful for debugging.

Tips and Tricks

- Every unique process (each server replica, each client, the RM, each LFD, the GFD) should have a dedicated terminal/console window so that it's easy to see its operation and output, while the process is running.
- The clients, the GFD and the RM all ideally run on a single separate machine.
- You can use the same client code for each of the clients, C1, C2 and C2, except that you want to label the clients uniquely, and each client should run independently of the other two (make sure that they don't share resources).
- You can use the same server code for both active replication and passive replication.
- LFDs should run on the same machine as the respective server replicas that they are heartbeating (e.g., LFD1 runs on the machine for replica S1 that it is heartbeating).
- All messages should be printed on the console windows of the specific processes. Every message sent to and from the servers, the clients, the LFDs, the GFD, and the RM should be printed to the console windows. For example, every time that server S1

receives a message from a client C1, the receipt of the request message from C1 to S1 must be visible on the console window of both S1 and C1. Similarly, the sending of the reply message back from S1 to C1 must be visible on the console window of both S1 and C1.

- The LFD's heartbeats (both sent and received) must be printed out on its console window, for every periodic heartbeat. For example, every time that LFD1 heartbeats server S1, the LFD1's console window must display the text, *[timestamp] [heartbeat_count] LFD1 sending heartbeat to S1*. Similarly, the receipt of the heartbeat message back from S1 should result in the display of the text, *[timestamp] [heartbeat_count] LFD1 receives heartbeat from S1.*

- Use a command-line parameter to set the LFD's configurable *heartbeat_freq*, and don't hard-code it into your LFD. Each LFD starts with *heartbeat_count=1*, when it first launches.

- Use a command-line parameter to set the primary replica's *checkpoint_freq* (in passive replication), and don't hard-code it into your application. The primary replica and the backup replicas start with *checkpoint_count=1*, when they first launch.

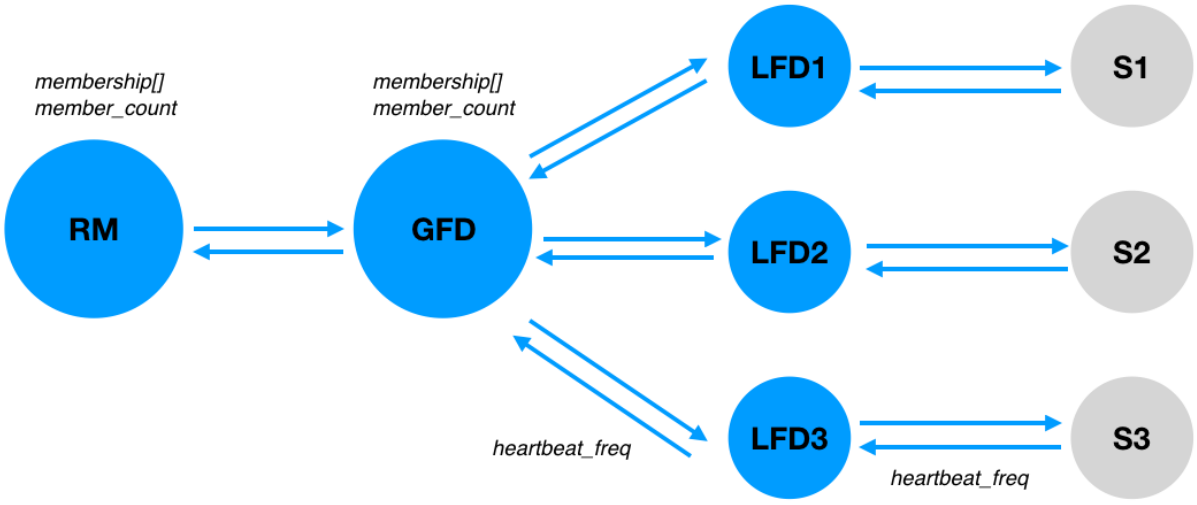
- With active replication, each server replica has (a) 3 TCP/IP connections, one each to C1, C2, and C3, which are used when the client communicates with the server, (b) 1 TCP/IP connection to its LFD, which is used for heartbeating, and (c) 2 TCP/IP connections to each of its fellow replicas (used only for initial state transfer, after which this "secondary channel" connection is not needed). All active replicas, regardless of whether or not they are primary or backup, need to open and maintain these connections, although some of these connections may not have much traffic on them.

- With passive replication, each server replica has (a) 3 TCP/IP connections, one each to C1, C2, and C3, which are used when the client communicates with the server and are used only with the primary replica, but not at the backup replicas, (b) 1 TCP/IP connection to its LFD, which is used for heartbeating, and (c) 2 TCP/IP connections to each of its fellow replicas (the "secondary channel" used only for periodic checkpointing). All passive replicas, regardless of whether or not they are primary or backup, need to open and maintain these connections, although some of these connections may not have much traffic on them.

- Each LFD has (a) *n* TCP/IP connections, one for every one of the *n* server replicas that it heartbeats, and (b) 1 TCP/IP connection to the GFD.

- The GFD has (a) *y* TCP/IP connections, one for every one of the *y* LFDs that it heartbeats, and (b) 1 TCP/IP connection to the RM.
- When each server replica is launched, it “registers” with its local LFD, for the heartbeating (of the server replica by its corresponding LFD) to commence. This means that the LFD is a known IP address and port on the machine for the server replica to initiate a TCP/IP connection with.
- When each LFD is launched, it “registers” with the GFD, for the heartbeating (of the LFD by the GFD) to commence. This means that the GFD is a known IP address and port in the system for each LFD to initiate a TCP/IP connection with.
- When the GFD is launched, it “registers” with the RM, for communicating membership changes. This means that the RM is a known IP address and port in the system for the GFD to initiate a TCP/IP connection with.
- Replicas can be automatically relaunched in multiple ways, one of which is to make the RM responsible for remotely launching the previously-failed replica, or by delegating that work to the respective LFD on that machine. This is an implementation choice.

Supplementary Illustrations



MILESTONE #1 EXPECTATIONS (5% of GRADE)

Goals

Get the client-server application running.
Demonstrate that the LFD works.
Support configurable heartbeat frequencies at the LFD.

> At this stage of the project, don't worry as yet about replication, the Replication Manager, the Global Fault Detector, duplicate detection, or about automating the clients' request messages in a loop.

Components

- 1 server replica (S1)
- 3 independent clients (not replicated)
- 1 LFD (LFD1)

Rubric

- (1) Launch the server replica, S1.
- (2) Launch the LFD1, on the same machine, with some default *heartbeat_freq*. It should start heartbeating the server replica periodically, and in a continuous loop. The periodic messages from LFD1→S1 must be visible and printed on the console window for the LFD1. The sending and receipt of these messages must be printed on the console window for server S1.
- (3) Launch the 3 independent clients, C1, C2 and C3, one at a time.
- (4) The 3 clients should start sending messages to the server replica, S1. You can manually send messages from each client.
- (5) The sending of the request messages from C1→S1 , C2→S1, C3→S1 must be printed on the console windows for each of the clients, C1, C2, and C3, respectively. The receipt of these request messages must also be visible on the console window for server S1.

- (6) The sending of the reply messages S1→C1, S1→C2, S1→C3 must be printed on the console windows for each of the clients, C1, C2, and C3, respectively. The sending of these reply messages must also be visible on the console window for server S1.
- (7) Show that the server S1 is "stateful," and that its state is being modified by the receipt of the clients' messages. On the console window, print the value of *my_state* before returning the reply to the client.
- (8) Try killing (Control-C) the replica, S1, and you should see a heartbeat fail (i.e., timeout expiration) at the LFD1. The LFD1 should be able to detect that the replica, S1, died. Print this failed-heartbeat message on the console window of LFD1.
- (9) Restart the whole system (repeat steps above) with a different *heartbeat_freq* for the LFD1.

MILESTONE #2 EXPECTATIONS (5% of GRADE)

Goals

Implement an actively-replicated server with 3 replicas.
Demonstrate that the clients' requests go to all 3 replicas, and that all 3 replicas respond.
Demonstrate that the clients suppress duplicate responses.
Demonstrate that the server continues functioning despite a single failed replica.

Components

- 3 active server replicas, S1, S2 and S3, ideally on 3 different distinct machines
- 3 LFDs, labeled LFD1, LFD2, LFD3, one for each of the respective replicas, e.g., LFD1 heartbeats S1. Note that server replica S1 needs to be on the same machine as LFD1, S2 on the same machine as LFD2, S3 on the same machine as LFD3.
- Each LFD is located on the same machine as the replica that it heartbeats.
- 3 independent clients C1, C2, and C3 (not replicated)
- 1 GFD, located on the same machine as the clients.

> At this stage of the project, don't worry as yet about passive replication, the Replication Manager, recovery, checkpointing, or about automating the clients' request messages in a loop.

Rubric

- (1) Launch the GFD. Its *member_count* = 0 and it prints the following text to its console: "*GFD: 0 members*"
- (2) Launch LFD1, LFD2, and LFD3, each on a separate machine, each with some default *heartbeat_freq*. Use the same frequency for all three of them. Each of the LFDs opens a TCP/IP connection with the GFD to communicate with (or "register its machine with") the GFD, so that the heartbeating can start between the GFD and LFDs. The replicas are not yet involved.
- (3) Launch the server replica S1 on machine 1. You should see heartbeat messages from LFD1→S1 being printed out on the console window for LFD1 and S1. After the first successful heartbeat, LFD1 sends a message (*"LFD1: add replica S1"*) to GFD to register the replica S1 as a member. GFD prints text to its console (*"GFD: 1 member: S1*"), and updates its *membership[]* and *member_count* to include the *replica_id* of replica S1.
- (4) Launch the server replica S2 on machine 2. You should see heartbeat messages from LFD2→S2 being printed out on the console window for LFD2 and S2. After the first successful heartbeat, LFD2 sends a message (*"LFD2: add replica S2"*) to GFD to register the replica S2 as a member. GFD prints text to its console (*"GFD: 2 members: S1, S2*"), and updates its *membership[]* and *member_count* to include the *replica_id* of replica S2.
- (5) Launch the server replica S3 on machine 3. You should see heartbeat messages from LFD3→S3 being printed out on the console window for LFD3 and S3. After the first successful heartbeat, LFD3 sends a message (*"LFD3: add replica S3"*) to GFD to register the replica S3 as a member. GFD prints text to its console (*"GFD: 3 members: S1, S2, S3*"), and updates its *membership[]* and *member_count* to include the *replica_id* of replica S3.
- (6) Launch the 3 clients, C1, C2, and C3, one at a time. Each client opens 3 TCP/IP connections, one to each replica.
- (7) The clients should start sending messages to the server replicas, with *request_num* numbers within the messages. You can manually send messages from each client.

- (8) The sending of the request messages from C1→S1, C1→S2, C1→S3 must be printed on the console window for C1. The receipt of these request messages must also be printed on the console window for S1, S2, and S3. Ditto for C2 and C3.
- (9) The sending and receipt of the reply messages, S1→C1, S2→C1, S3→C1 must be visible on the console windows for C1 and on each of the replicas, S1, S2, and S3, respectively. Ditto for C2 and C3.
- **(10) At this stage, you have a functioning fault-free distributed active-replication system with the ability to tolerate 2 simultaneous faults.**
- (11) Try injecting a crash fault by killing replica S1 (Control-C). You should see the heartbeat fail (i.e., timeout expiration) at LFD1. Print this failed-heartbeat message on the console window of LFD1.
- (12) LFD1 sends a membership-change message (*"LFD1: delete replica S1"*) to the GFD. The result is that the GFD prints text to its console, *"GFD: 2 members: S2, S3.*"
- (13) The clients should still be sending requests to, and receiving replies from, the remaining server replicas, S2 and S3. You can manually send messages from each client. The sending and receipt of the requests C1→S2, C1→S3 must be printed on the console window for C1, S2, and S3. Ditto for C2 and C3.
- (14) The sending and receipt of the replies S2→C1, S3→C1 must be visible on the console window for C1, S2, and S3. Ditto for C2 and C3.
- (15) Throughout the set of steps above, the client should detect duplicate replies and print them to the console using the information in the tuple <*client_id, replica_id, request_num*>. For example, C1 should detect that <*C1, S2, 134, reply*> *and* <*C1, S3, 134, reply*> are duplicate replies in response to the associated requests C1→S2, C1→S3, both with *request_num* 134. This detection of duplicates should be printed to C1's console window as: *"request_num 134: Discarded duplicate reply from S3."*

MILESTONE #3 EXPECTATIONS (5% of Grade)

Goals

Implement a passively-replicated server with 3 replicas.
Demonstrate that the clients' requests go to the primary replica, and that only the primary replica responds.

Demonstrate that the primary replica sends checkpoints to the backup replicas.
Demonstrate that the server continues functioning despite a single failed **backup** replica.

Components

- 3 passive server replicas, with S1 (initially the primary replica), S2 and S3, ideally on 3 different distinct machines
- 3 LFDs, labeled LFD1, LFD2, LFD3, one for each of the respective replicas, e.g., LFD1 heartbeats S1. Note that server replica S1 needs to be on the same machine as LFD1, S2 on the same machine as LFD2, S3 on the same machine as LFD3.
- Each LFD is located on the same machine as the replica that it heartbeats.
- 3 independent clients C1, C2, and C3 (not replicated)
- 1 GFD, located on the same machine as the clients.

> At this stage of the project, don't worry as yet about recovery, the Replication Manager, or about automating the clients' request messages in a loop.

Rubric

- (1) Launch the GFD. Its *member_count* = 0 and it prints the following text to its console: "*GFD: 0 members*"

- (2) Launch LFD1, LFD2, and LFD3, each on a separate machine, each with some default *heartbeat_freq*. Use the same frequency for all three of them. Each of the LFDs opens a TCP/IP connection with the GFD to communicate with (or "register its machine with") the GFD, so that the heartbeating can start between the GFD and LFDs. The replicas are not yet involved.

- (3) Launch the server replica S1 on machine 1. You should see heartbeat messages from LFD1→S1 being printed out on the console window for LFD1 and S1. After the first successful heartbeat, LFD1 sends a message (*"LFD1: add replica S1"*) to GFD to register the replica S1 as a member. GFD prints text to its console (*"GFD: 1 member: S1*"), and updates *membership[]* and *member_count* to include the *replica_id* of replica S1.

- (4) Launch the server replica S2 on machine 2. You should see heartbeat messages from LFD2→S2 being printed out on the console window for LFD2 and S2. After the first successful heartbeat, LFD2 sends a message (*"LFD2: add replica S2"*) to GFD to

register the replica S2 as a member. GFD prints text to its console (*"GFD: 2 members: S1, S2"*), and updates *membership[]* and *member_count* to include the *replica_id* of replica S2.

- (5) Launch the server replica S3 on machine 3. You should see heartbeat messages from LFD3→S3 being printed out on the console window for LFD3 and S3. After the first successful heartbeat, LFD3 sends a message (*"LFD3: add replica S3"*) to GFD to register the replica S3 as a member. GFD prints text to its console (*"GFD: 3 members: S1, S2, S3"*), and updates *membership[]* and *member_count* to include the *replica_id* of replica S3.
- (6) Replica S1 opens 2 TCP/IP connections, one to each backup replica, S2 and S3. This is the "secondary channel" for sending checkpoints.
- (7) Launch the 3 clients, C1, C2, and C3, one at a time. Each client opens 3 TCP/IP connections, one to each replica.
- (8) The clients should start sending messages to the primary server replica S1, with *request_num* numbers within the messages. You can manually send messages from each client.
- (9) The sending and receipt of the requests from C1→S1 must be visible on the console window for C1 and S1. Ditto for C2 and C3 (for the requests that they respectively send to S1). Replicas S2 and S3 are not doing anything now. They are essentially idling. 😴
- (10) The sending and receipt of the reply S1→C1 must be visible on the console windows for C1 and S1. Ditto for the replies from S1 to C2 and C3.
- (11) Periodically, at a *checkpoint_freq*, replica S1 takes a checkpoint of its state and sends a checkpoint message (containing the current value of *my_state* and the current value of *checkpoint_count*) to backup replicas, S2 and S3. S1 also increments the value of *checkpoint_count* after sending the checkpoint message. The console window of S1 displays the sending of this checkpoint message.
- (12) In turn, backup replicas, S2 and S3, display the received checkpoint messages on their console windows, and set their own internal state to the value of *my_state* and also update their local value of *checkpoint_count* based on the one received in the checkpoint message.

- **(13) At this stage, you have a functioning fault-free distributed passive-replication system with the ability to tolerate 2 simultaneous faults. Let's inject a crash-fault now!**
- (14) Try injecting a crash fault by killing replica S2 (Control-C). You should see the heartbeat fail (i.e., timeout expiration) at LFD2. Print this failed-heartbeat message on the console window of LFD2.
- (15) LFD2 sends a membership-change message (*"LFD2: delete replica S2"*) to the GFD. GFD prints text to its console: *"GFD: 2 members: S1, S3."*
- (16) The clients should still be sending requests to, and receiving replies from, the primary server replica, S1. You can manually send messages from each client. The sending and receipt of the request C1→S1, must be visible on the console window for C1 and S1. Ditto for requests sent by C2 and C3.
- (17) The sending and receipt of the reply S1→C1 must be visible on the console window for C1 and S1. Ditto for replies sent by S1 to C2 and C3.
- (18) Throughout the set of steps above, the sending and receipt of the checkpoints should be printed on the console windows of S1, S2 (when it is alive), and S3. Both the sending of the checkpoint as well as the receipt of the checkpoint must be visible. In all cases, the value of *my_state* and *checkpoint_num* should be printed to the console windows of S1, S2, and S3, to show the ongoing checkpointing in action.

MILESTONE #4 EXPECTATIONS (5% of Grade)

Goals

Implement an actively-replicated server with 3 replicas, with sustained operation in the fault-free case, the single-fault case, and the recovery-from-fault case.
Implement a passively-replicated server with 3 replicas, with configurable checkpointing, with sustained operation in the fault-free case, the single-fault case (backup fails), the single-fault case (the primary fails), the recovery-from-primary-failing case (primary fails and a new primary is elected), and the recovery-from-backup-failing case (backup fails and is relaunched).
Demonstrate the RM, GFD, and LFDs working in concert as the fault-tolerance infrastructure.
Demonstrate manual recovery (bring up the failed replicas manually) with checkpointing.

Components

- **Configuration #1**: 3 active server replicas, S1, S2 and S3, ideally on 3 different distinct machines
- **Configuration #2:** 3 passive server replicas, S1 (initially the primary replica), S2 and S3, ideally on 3 different distinct machines
- In each configuration: 3 LFDs, labeled LFD1, LFD2, LFD3, one for each of the respective replicas, e.g., LFD1 heartbeats S1. Note that server replica S1 needs to be on the same machine as LFD1, S2 on the same machine as LFD2, S3 on the same machine as LFD3.
- In each configuration: Each LFD is located on the same machine as the replica that it heartbeats.
- In each configuration: 3 independent clients C1, C2, and C3 (not replicated)
- In each configuration: 1 RM and 1 GFD, both located on the same machine as the clients.

> At this stage of the project, don't worry as yet about automated recovery, varying the checkpointing frequency, varying the heartbeat frequency, or about automating the clients' request messages in a loop.

Rubric for Configuration #1

- (1) Launch the RM. Its member_count = 0, and it prints the following text to its console: *"RM: 0 members"*
- (2) Launch the GFD. Its *member_count* = 0 and it prints the following text to its console: "*GFD: 0 members*"
- (3) The GFD opens a TCP/IP connection to the RM, to communicate with (or "register with") the RM. The GFD should notify the RM that it has 0 members in its group. The RM indicates its knowledge of the system membership by printing the following text to its console, *"RM: 0 members."*
- (4) Launch LFD1, LFD2, and LFD3, each on a separate machine, each with some default *heartbeat_freq*. Use the same frequency for all three of them. Each of the LFDs opens a TCP/IP connection with the GFD to communicate with (or "register its machine with") the GFD, so that the heartbeating can start between the GFD and LFDs. The replicas are not yet involved.

- **(5) At this stage, you have a functioning fault-tolerance infrastructure consisting of the RM, GFD, and LFDs. This system is ready to take on the application, and protect it from faults.**
- (6) Launch the server replica S1 on machine 1. You should see heartbeat messages from LFD1→S1 being printed out on the console window for LFD1 and S1. After the first successful heartbeat, LFD1 sends a message (*"LFD1: add replica S1"*) to GFD to register the replica S1 as a member. GFD prints text to its console (*"GFD: 1 member: S1"*), and updates its *membership[]* and *member_count* to include the *replica_id* of replica S1. GFD sends the membership information to RM, and RM prints text to its console (*"RM: 1 member: S1"*), and updates its *membership[]* and *member_count* to include the *replica_id* of replica S1.
- (7) Launch the server replica S2 on machine 2. You should see heartbeat messages from LFD2→S2 being printed out on the console window for LFD2 and S2. After the first successful heartbeat, LFD2 sends a message (*"LFD2: add replica S2"*) to GFD to register the replica S2 as a member. GFD prints text to its console (*"GFD: 2 members: S1, S2"*), and updates its *membership[]* and *member_count* to include the *replica_id* of replica S2. GFD sends the membership information to RM, and RM prints text to its console (*"RM: 2 members: S1, S2"*), and updates its *membership[]* and *member_count* to include the *replica_id* of replica S2.
- (8) Launch the server replica S3 on machine 3. You should see heartbeat messages from LFD3→S3 being printed out on the console window for LFD3 and S3. After the first successful heartbeat, LFD3 sends a message (*"LFD3: add replica S3"*) to GFD to register the replica S3 as a member. GFD prints text to its console (*"GFD: 3 members: S1, S2, S3"*), and updates its *membership[]* and *member_count* to include the *replica_id* of replica S3. GFD sends the membership information to RM, and RM prints text to its console (*"RM: 3 members: S1, S2, S3"*), and updates its *membership[]* and *member_count* to include the *replica_id* of replica S3.
- (9) Replica S1 opens 2 TCP/IP connections, one to each backup replica, S2 and S3. This is the "secondary channel" for sending checkpoints to a new active replica.
- (10) The clients should start sending messages to the server replicas, with *request_num* numbers within the messages. You can manually send messages from each client.

- (11) The sending of the request messages from C1→S1, C1→S2, C1→S3 must be printed on the console window for C1. The receipt of these request messages must also be printed on the console window for S1, S2, and S3. Ditto for C2 and C3.
- (12) The sending and receipt of the reply messages, S1→C1, S2→C1, S3→C1 must be visible on the console windows for C1 and on each of the replicas, S1, S2, and S3, respectively. Ditto for C2 and C3.
- **(13) At this stage, you have a functioning fault-free distributed active-replication system with the ability to tolerate 2 simultaneous faults, and with the complete fault-tolerance infrastructure consisting of the RM, GFD, and LFDs. Let's inject a crash fault now!**
- (14) Now, kill (Control-C) an active replica, say, S2. The heartbeat LFD2→S2 should timeout and fail. Print this failed-heartbeat message on the console window of LFD1.
- (15) LFD2 sends a membership-change message (*"LFD2: delete replica S2"*) to the GFD. The result is that the GFD prints text to its console, *"GFD: 2 members: S1, S3.*"
- (16) The clients should still be sending requests to, and receiving replies from, the remaining server replicas, S1 and S3. You can manually send messages from each client. The sending and receipt of the requests C1→S1, C1→S3 must be printed on the console window for C1, S1, and S3. Ditto for C2 and C3.
- (17) The sending and receipt of the replies S1→C1, S3→C1 must be visible on the console window for C1, S2, and S3. Ditto for C2 and C3.
- (18) Throughout the set of steps above (both fault-free and fault-injected), the client should detect duplicate replies and print them to the console using the information in the tuple <*client_id, replica_id, request_num*>. For example, C1 should detect that <*C1, S1, 134, reply*> *and* <*C1, S3, 134, reply*> are duplicate replies in response to the associated requests C1→S1, C1→S3, both with *request_num* 134. This detection of duplicates should be printed to C1's console window as: *"request_num 134: Discarded duplicate reply from S3."*
- **(19) Let's recover the failed replica now!**
- (20) Manually re-launch active replica S2. It should re-register with LFD2, and LFD2 should re-communicate the existence of S2 to the GFD, which should notify the RM. On the console windows of LFD2, GFD, and RM, we should see the text displayed that

indicates the re-launch of S2. The text on the console windows is identical to the text that you would see for the initial launch of replica S2. Basically, step (7) should take place all over again.

- (21) The active replicas S1 and S3 should continue processing normal client requests from C1, C2 and C3. However, replica S2 should not process any requests until it is ready to do so. Essentially, with replica S2, *i_am_ready = 0* at this point. S2 is considered ready (to receive client requests) only when it has received the latest checkpoint from one of the other active replicas. However, the new S2 should continue to receive and log incoming request numbers. S2 essentially updates the *high_watermark_request_num[]* with every new request that it sees from C1, C2, and C3. The newborn replica S2 cannot reply to the clients' messages because it does not have the correct state from the existing and healthy active replicas S1 and S2.

Rubric for Configuration #2

- Mimic the sequence for active replication, except that you need to show the failure of a backup replica, and the recovery of the backup replica.
- Mimic the sequence for active replication, except that you need to show the failure of the primary replica, and the reelection of a new primary replica.

MILESTONE #5 EXPECTATIONS (5% of Grade)

Goals

This is similar to Milestone #4, except that you're going to test with two faults injected, one after the other.

Demo Scenarios: Active Replication

Make sure that you test your actively-replicated application in all of these scenarios. You should make sure that you have run your system through these test scenarios. Each of these is a possible scenario that you will be asked to run at the final demo.

- **Bootstrapping**: The fault-tolerance infrastructure (the combination of RM, GFD, LFDs) initializes, and starts up for the first time in a clean state. This looks identical for both active and passive replication.
- **Steady-state v1**: The application runs normally (we now add S1, S2, S3), with no server faults, and with the server replicas protected by the fault-tolerance infrastructure.
- **Steady-state v2**: The application runs normally (we now add C1, C1, C3), with no server faults, and with the server replicas protected by the fault-tolerance infrastructure.
- **Fault-injected**: We will kill a replica. We may kill any single replica, chosen at random.
- **Fault-recovered**: We will recover the replica that we just killed, and ensure that it is operational again.
- **Fault-reinjected**: We will kill the active replica that we just recovered, to prove that everything continues to operate. We should be down to two active replicas.
- **Second-fault-injected**: We will kill a second active replica. At this stage, only one active replica should be left in the system, and it should be running just fine.

Demo Scenarios: Passive Replication

Make sure that you test your passive-replicated application in all of these scenarios. You should make sure that you have run your system through these test scenarios. Each of these is a possible scenario that you will be asked to run at the final demo.

- **Bootstrapping**: The fault-tolerance infrastructure (the combination of RM, GFD, LFDs) initializes, and starts up for the first time in a clean state. This is identical for both active and passive replication.
- **Steady-state v1**: The application runs normally (we now add S1, S2, S3), with no server faults, and with the server replicas protected by the fault-tolerance infrastructure. Designate one of the replicas as the primary replica. We should be able to designate any replica as the primary replica. Basically, don’t hard-code which replica is the primary.
- **Steady-state v2**: The application runs normally (we now add C1, C1, C3), with no server faults, and with the server replicas protected by the fault-tolerance infrastructure.

We should see the checkpoints happen quite normally between the primary and the backup replicas.

- **Fault-injected**: We will kill the primary replica. We should see a new primary replica get elected, and we should see checkpointing continue normally, and the clients receive the responses. We are now down to two passive replicas.
- **Fault-recovered**: We will recover the killed ex-primary replica. It should come back up as a backup replica now. It should start receiving checkpoints from the new primary replica, and proceed to operate normally. We should now have three passive replicas again.
- **Fault-reinjected**: We will kill the backup replica that we just recovered, to prove that everything continues to operate. We should be down to two passive replicas.
- **Second-fault-injected**: We will kill a second backup replica. At this stage, only one passive replica (the primary replica) should be left in the system, and it should be running just fine.

PROJECT-MANAGEMENT EXPECTATIONS

Take turns being the project manager

There are 5 people in each project team. Each of you will play the role of project manager for a specific milestone. It is up to each team to decide who is the project manager for which milestone.

When you are the project manager for a specific milestone, your responsibilities include:

- **Smooth and successful milestone demo:** The demo for the milestone will be presented by you, on behalf of the entire team.
- **Explanations for the milestone demo:** The rest of the team will be present, but you will explain how the demo works, how the components work, what is the architecture, what system decisions your team made, what assumptions your team made, how your team has successfully met the milestone, and how your overall system works.
- **Q&A about the milestone demo:** You are also responsible for answering questions about any limitations of your approach.

FREQUENTLY ASKED QUESTIONS

What programming language should I write the application in?

The programming language does not matter. Choose the language that you are most comfortable with.

Are there resources for me to learn network programming in Java?

- Overview of networking (the basics, ports, sockets, etc.). https://docs.oracle.com/javase/tutorial/networking/overview/index.html
- Network programming in Java. https://www.tutorialspoint.com/java/java_networking.htm
- Another network programming tutorial for Java. https://www.infoworld.com/article/2853780/socket-programming-for-scalable-systems.html
- Slides from a course on network programming in Java. http://akira.ruc.dk/~keld/teaching/CAN_f13/Slides/pdf/JavaNet.pdf
- Udemy course on network programming in Java. https://www.udemy.com/course/network-programming-java-network-programming-nio-tcpip-sockets/ — Disclaimer: I don't know if this course is good or not. If you want to take this course to get ready for the class project, I will pay the Udemy fees for it. It's a 6-hour course, though, which is why I don't want to burden anyone with it. However, please let me know if you are interested, and I can set up a PayPal account to pay for the Udemy course.

Are there resources for me to learn network programming in Python?

- https://realpython.com/python-sockets/ — appears to be a tutorial on using Python libraries for sockets.
- https://docs.python.org/3/library/socket.html — low-level networking socket interfaces in Python.
- https://www.udemy.com/course/python-network-programming-tcpip-socket-programming/ — Disclaimer: I don't know if this course is good or not. If you want to take this course to get ready for the class project, I will pay the Udemy fees for it. It's a 4-hour course, though, which is why I don't want to burden anyone with it. However, please let me know if you are interested, and I can set up a PayPal account to pay for the Udemy course.

What should the application do? Is there a specific functionality we should have?

The application's functionality does not matter. The only part that matters is that there are multiple independent clients and that there is a server that has state in it. Each client's requests should change the state in the server. The state can be as simple as a variable, x, that changes each time a client's request is received.

Can I use the same client for all 3 clients?

Absolutely, as long as their messages are uniquely identified, and it's clear which client is communicating with the server at any point in time.

What user interface do I need for the application?

You don't need a fancy UI for the application. We need to see all of the messages being printed out on the console/terminal windows, to see all of the communication happening in the system. So, your demo can consist of a number of console/terminal windows, each of them running a different component.

What do I do with all of the components that are single points of failure?

Put the unreplicated single RM, the unreplicated single GFD, and the 3 clients on the same machine, to make things easier to manage.

What should run on each machine that hosts a server replica?

On each machine that has a server replica, you should run a LFD that heart-beats its local server replica.

Can I use a different implementation approach from the one described here?

Absolutely, as long as you meet the goals of the demo, and can show fault-tolerance, recovery, checkpointing, ordering, duplicate detection, fault-detection, etc., in action.

Can I combine the RM and GFD into one?

It's not ideal, but if you choose to, absolutely! Combining the two does not take away from the fault-tolerance of the system. In commercial systems, the RM and GFD have different roles, which is why they are introduced as different components.

What do you mean by trade-offs in the system?

You will likely be making implementation decisions that trade-off consistency vs. performance. For example, you might choose to use a specific implementation to ensure consistency, but it might be a blocking implementation that then affects latency.

Should the clients be automated, and running in a loop?

For the final demo, yes, so that the entire application is continuing to run during the demo, and so that we can focus our attention (during the demo) on killing and restarting the server replicas while the clients continue running full tilt.

Should the RM launch everything?

In commercial systems, the RM launches the GFD, the LFDs, and the server replicas. In your implementation, you can choose to launch these manually, if you like. However, even in your implementation, the RM must have a mode of operation when it automatically re-launches dead replicas and keeps the system running with 3 replicas all the time.

EXTRA-CREDIT OPTIONS (5% of Grade)

Goals

This serves to provide an opportunity for individual students to implement and demonstrate additional functionality after Milestone #4, in order to showcase some unique capability for their projects.

Pick **only one** of the options below to aim for the extra-credit grade.

Extra-Credit Option #1: Black-Box Failure Diagnosis

In this case, you will inject an artificial performance-slowdown fault, e.g., running out of memory, or a CPU-intensive task. You will show how black-box metrics can be used to detect the fault, and diagnose the root-cause.

- **Black-box instrumentation**: Implement a way to gather and report black-box metrics periodically from the OS (e.g., CPU usage, memory usage, disk usage). Show the data in some visual way, e.g., graphs of the data.
- **Black-box "normal" behavior signature:**: Derive the black-box signature of normal behavior under fault-free conditions.
- **Black-box anomaly detection**: Demonstrate that the performance-slowdown fault manifests on the black-box metrics that you are collecting, and show how anomaly detection can help to diagnose the root-cause of the problem.
- **Tips:** Don't attempt too many kinds of faults. Pick a simple performance-slowdown (or fail-slow) fault. Pick a performance fault that is likely to manifest on black-box metrics (e.g., CPU usage, disk usage, memory usage, bandwidth usage).

Extra-Credit Option #2: White-Box Failure Diagnosis

In this case, you will inject an artificial performance-slowdown fault, e.g., running out of memory, or a CPU-intensive task. You will show how white-box metrics can be used to detect the fault, and diagnose the root-cause. You will use path-based analysis with latency (the time taken for a message to be transmitted from one process to another, across the network).

- **White-box instrumentation**: Implement a way to gather and report white-box metrics periodically from the application, and to show paths of the Service-Level Objectives, in terms of latency. For example, show the latency breakdown in sending end-to-end messages, and the latency between messages exchanged between clients and server replicas, between the LFDs and the server replicas, between the LFDs and the GFD, and between the GFD and the RM. You want the latency (averages, min, max) to be measured and reported between every pair of processes in the system.
- **White-box "normal" path signature:**: Derive the white-box signature of normal path behavior under fault-free conditions.
- **White-box anomaly detection**: Demonstrate that the performance-slowdown fault manifests on the white-box latency metrics that you are collecting, and show how anomaly detection can help to diagnose the root-cause of the problem.

- **Tips:** Don’t attempt too many kinds of faults. Pick a simple performance-slowdown (or fail-slow) fault. Pick a performance fault that is likely to manifest on white-box latency metrics.