ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

PBFT PROOF OF CONCEPT WITH SARL

Progetto realizzato nel corso di:
Sistemi Autonomi

Progetto di:
LUDOVICO DE NITTIS

ANNO ACCADEMICO 2016–2017

Intro

This project is a proof of concept of the Practical Byzantine Fault Tolerance.

The PBFT gives the ability to independent distributed nodes to come to an agreement on the state of the system, tolerating a certain amount of node failures and/or manipulated messages.

The algorithm, to work as expected, requires that number of faulty/malicious nodes can't exceed 1/3 of the total nodes number (at a given time).

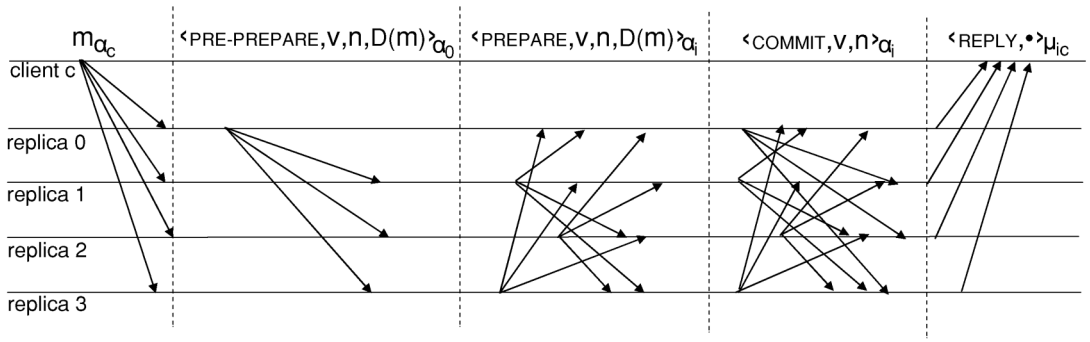
Each request of agreement is executed with a five phases process:

- The Client multicast the request message to every available nodes.
- The Primary propagate the request to the Replicas assigning a sequence number to it.
- The Backups multicast a message like the one received from the Primary, declaring that they accepted the received assignment.
- Another message is multicasted from all replicas that ensure a total order for requests, even across view changes.
- The Replicas execute the request and send the result back to the Client that will wait `f + 1` valid replies.

Language

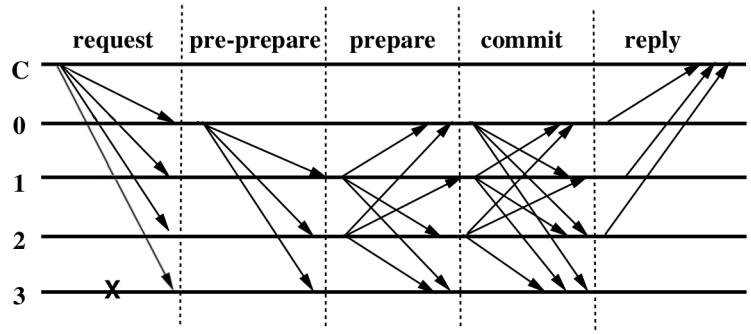
For the implementation SARL, a General-purpose Agent-Oriented Programming Language, has been used.

Normal case operation



Normal_case

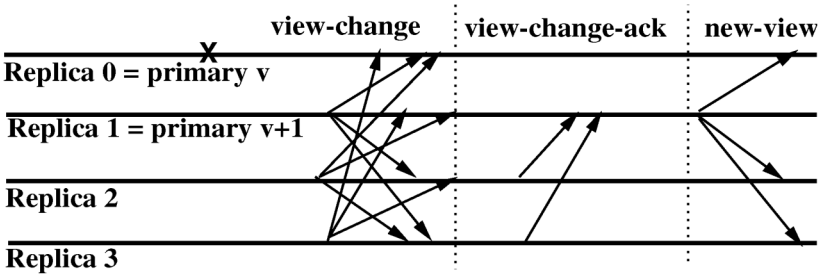
Faulty replica



Faulty_replica

Faulty primary

In case the replicas suspects the primary is faulty they trigger a view-change.



Glossary

- m = client's request message
- v = view
- D(m) = digest of m
- n = sequence number
- i = node id
- <m>u = point-to-point message that contain a single MAC
- <m>a = multicast message with a vector of MACs called authenticator
- <m>o = multicast message signed with the private key of the sender

Summary of the implementation

`Initializer.sarl` is an auxiliary agent with the only purpose of initializing the system with the specified number of replicas and one client.

`Node.sarl` is a base agent extended by both `Client.sarl` and `Replica.sarl`.

`SkillNodeHandling.sarl` is a skill that implements the capacity of `KeysHandling.sarl`.

`Events.sarl` contains all the events/messages that can happen in the system.

`CryptoUtil.java` is the only Java class and it let the SARL agents to perform all the required crypto functions like encrypting/decrypting, signing, ecc...

Initializer

The `initializer.sarl` is the starting agent of the system. His only purpose is to spawn all the required agents (one client and n replicas). This is also where we can configure the system: changing the number of allowed faulty agents, the ones actual faulty and the viewID. A more in-depth description on how to configure them is explained below.

Before the creation of the requested nodes (agents), replicas + 1 pair of private-public keys will be generated and stored in a temporary array.

At spawn time the nodes will receive, as an argument, their personal private key and an array with the public keys of all the other nodes in the system. In this way the agents will be able to authenticate their communications and prevent possible attacks like an active man-in-the-middle.

The assumption here is that the initializer can not be faulty, otherwise the system will not be able to even start correctly.

Cryptographic utilities

All the required crypto utilities functions have been implemented in a java class called `CryptoUtil.java`. It includes:

- generateKeyPair: It generates an RSA 2048 bit key pair.
- encrypt: Encrypts a String message with the given public key.
- sign: Given a New-key event and a private key, it will return the event's signature

- decrypt: The opposite of encrypt, with a private key and an encrypted message it will decrypt it and return his plain text
- verify: Given the received New-key event, the received signature and the known public key, this function will check if the sender used the right private key. E.g. if the received message has not been tampered.

KeysHandling and SkillNodeHandling

All the functions about the keys management, like refreshing the MAC key pairs or calculating the MAC for a given message, has been declared in the `KeysHandling` capacity and implemented in the `SkillNodeHandling` skill.

A Capacity is the specification of a collection of Actions (in our case the actions are the functions inside `KeysHandling`). It is used to specify what an Agent can do and what behavior is required for its execution.

A Skill is a collections of Actions implementing a Capacity as described in its specification.

"An Agent can dynamically evolve by acquiring (learning) new Capacities, and it can also dynamically change the Skill associated with a given Capacity. Acquiring a new Capacity enables an Agent to get access to new behaviors. This provides Agents with a self-adaptation mechanism that allows them to dynamically change their architecture according to their current needs and goals."

The `SkillNodeHandling` is used by both Client and Replicas. They learn this skill when they are initialized. Even if not used in the current implementation, these keys handling functions have been designed as Capacity and Skill so that we can easily change them at runtime, teaching an Agent new functions or replacing their implementation when needed (for example when they change their role).

Node

`Node` is a base Agent. Both `Client` and `Replica` extend from it.

It has all the common things between `Client` and `Replica`. For example a few variables and also the `NewKey` handler.

In this way it has been possible to avoid duplicating code between different Agents.

Client

A Client is the Agent that starts the distributed task sending a new Request to the replicas and waiting until it receives f + 1 valid replies.

When the Client is initialized it takes three parameters:

- `f`: The number of allowed faulty nodes
- `publicKeys`: An array list with the public keys of the replicas
- `privateKey`: The personal private key

Replica

A replica (backup) in an Agent that has the goal of executing the received requests. It will hop between stages called: PrePrepared, Prepared and Committed. These types of stages ensures that the primary assigns a valid sequence number to the requests and also that all the replicas agrees on how and what tasks they need to perform.

When the Replica is initialized it takes six parameters:

- `nodeID`: The ID of this replica. It is an integer number that can start from `1`.
- `f`: The number of allowed faulty nodes
- `viewID`: The starting view ID
- `publicKeys`: An array list with the public keys of the client and the other replicas
- `privateKey`: The personal private key
- `faulty`: This is a boolean variable. If is true the node will act as if it is faulty.

Keys exchange

All the messages between the nodes needs to be encrypted and/or signed. Using just a Public-key cryptography would have been enough, but the problem is that the encryption and decryption is very expensive. To avoid a performance hit, the Public-key cryptography is used in combination with the Message authentication code.

When the Agents are initialized the first thing they do is sending the <NewKey, i, ... , {kj, i}C j , ... , t>o message, and schedule to repeat it every minute. This message is signed with the sender private key and every generated MAC keys is encrypted with the corresponding receiver's public key.

From now on the client and replicas can communicate signing the messages with the faster MAC (is a symmetric encryption) instead of using the Public Key Cryptography, which will be used again only for refreshing the MAC keys with another NewKey message.

How to get started

The project has a maven build automation system that takes care of the project dependencies.

Apparently Xtext is not yet compatible with Eclipse Photon, so the recommended way of starting this project is by downloading the latest version of SARL, importing this project and executing `Initializer.sarl` as a SARL Agent (Run as -> SARL Agent).

Alternatively it is possible to launch it from the CLI. Open a terminal in the project root directory and simply execute:

```
mvn clean package exec:java -Dexec.mainClass="io.janusproject.Boot"
-Dexec.args=com.sarl.pbft.Initializer
```

Configuration

There are three variables in `Initialized.sarl` that can be adjusted to configure the system:

- `f`: the number of allowed faulty nodes
- `faultyNodes`: the actual number of faulty nodes
- `viewID`: the starting view

The replicas number will be calculated as `replicas = 3 * f + 1`.

So if `faultyNodes` will be set at a value less or equal to `f` the algorithm will work as expected. Otherwise it will hang because there will be more faulty nodes than how the system can tolerate.

Create a faulty primary replica

The primary is chosen as `(viewID % replicas) + 1` and when we set that we want `x` faulty nodes, at creation time the first `x` will have the flag `faulty` set at true. Knowing that, if we change the `viewID` in a way that `(viewID % replicas) + 1 = 1`, with just one faulty node we will be sure that the faulty one will be the primary. Example:

```
f = 1
faultyNodes = 1
viewID = 16L
```

```
replicas = 3 * f + 1 = 4
primary = (viewID % replicas) + 1 = (16 % 4) + 1 = 1
```

Double faulty primaries

The pbft algorithm states that after a New-View message, the view is incremented of one. So it is possible to create a scenario where the first primary is faulty, and after a View-Change the new primary will be faulty as well. Example:

```
f = 2
faultyNodes = 2
viewID = 14L
```

TODO

The mechanism that prevents message logs from growing without bound called Garbage Collection has not been implemented.