

Task Organizer: Project report

Leonardo Cecchelli
Filippo Guggino
Riccardo Xefraj

March 2021

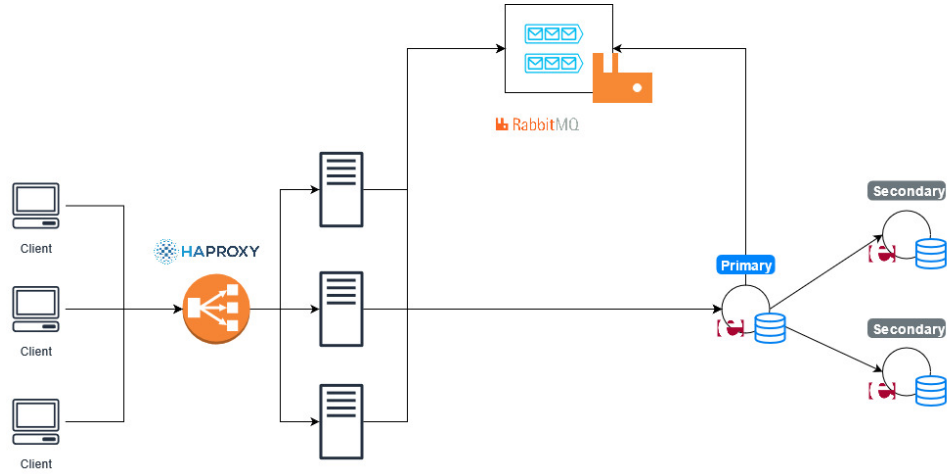
Abstract

Task Organizer is a web application whose goal is to deliver a task manager system for projects that embrace the Agile development approach. With this webapp a group of developers can create a board where they can save and organize all the tasks that are part of their project. They can create and move tasks to different stage of the Agile pipeline, so that each member can see the progress of the entire project.

1 Introduction

Since we have been using Trello for some university projects, we decided to make our own version of it but focused on just the Agile Development workflow. Each user can create or select an existing board, which is the place where all tasks will be stored and organized. Users accessing to a specific board can create or move tasks inside of it, adding all the necessary details like expiration date, a description, a type etc. At the moment we implemented four steps of the Agile pipeline, which are the following: Backlog, Doing, Quality Check, Done. The Backlog stage is where all new tasks are stored, they will wait there until some user will approve them by moving in the next stage. The Doing stage instead is where all tasks that are currently under work are saved, once finished they are moved to the Quality Check stage where some user will check that the results meet the initial requirements. Finally, once a task has been quality checked, it is moved to the Done stage where it is stored to keep track of the completed part of the project.

The overall architecture is composed by 3 key elements: the web server modules, the Erlang nodes that manages the database and RabbitMQ component for managing queues. Each one of these modules will be analysed in the next sections of the paper.



Finally, our main goals were to have data redundancy to prevent possible loss of data and we also wanted at the same time to guarantee consistency between the data that is visualized from the users. Since multiple users will access the same board at the same time, it is important that the state of this one is updated for all clients otherwise there will be some misunderstandings between the members of a team working on the same project.

2 Web Server module

This module was built using JSP pages, Servlets and Jinterface technologies, deployed then using Tomcat server instance. We will have a look at how each of this component has been implemented in the following paragraphs.

2.1 JSP Pages

In this paragraph we will analyze the JSP pages that the web server offers. In the specific we have two different pages that a client can request: `index.jsp` and `board.jsp`.

The first page `index.jsp` provides the user two forms, in the first one the user can select an available page while in the second one he can create a new one, in both cases once the submit button is pressed the user is redirected to the selected/created board.

The second page `board.jsp` is of course the main page where all tasks are grouped under the four stages. When the page is loaded parameters regarding the name of the board, the current date and the tasks are retrieved from the database using a servlet called `BoardServlet`, whose implementation will be explained in the next paragraph. In general, all tasks added to the board html using scriptlets, during this process the expiration date of the task is compared with the actual date since expiring one will be highlighted in red to attract user's attention. At the bottom of the page, we can find two forms, one is for creating a new task, which as we previously said will be added to the Backlog stage. The other form instead provides the possibility to move a task from a stage to another, the user simply has to put the name of the task and the stage to which he wants the task to be moved. Please note that two tasks belonging to the same board cannot have the same title.

2.2 Servlets and Jinterface

In this project we used 4 different servlets: `BoardServlet`, `CreateServlet`, `createBoard`, `MoveServlet`, and `WelcomeServlet`.

2.2.1 WelcomeServlet

This Servlet intercepts any GET request sent by the client at the url `/welcome/`. When activated it requests the list of all boards that are present in the database and set a specific parameter, finally it forwards the request to the `welcome.jsp` page.

2.2.2 BoardServlet

This servlet retrieves the name of the board that has been selected by the user, then requests all tasks that are present inside of it. It will receive an update package containing the tasks already divided in the corresponding stages. The servlet then creates 4 lists (one for each stage) and will set them as session attributes so that the scriptlets of the board page can retrieve them and print the corresponding html elements. Once the servlet has set these attributes it will forward the request to `/board.jsp`.

2.2.3 CreateServlet

This servlet is the one in charge of sending the newly created task information to the database system. It is activated by the form action inside the board page, therefore once started it will retrieve the parameters of the task from the GET request parameters. The parameters needed are 6, in the specific we have the task name, the type, the description, the expiration date, the creator of the task and the board name. After getting the parameters the servlet will create a new Task object and send it together with the board name to the database by using the **MessageManager** function called **sendCreateTask**. Finally, it will redirect back the client to the board page.

2.2.4 MoveServlet

The **MoveServlet** will send an update to the database since one task has changed stage. The behaviour is pretty like the previous servlet, although the parameters here are just 2: the board title and the new updated stage. In this case we use a different function called **sendMoveTask** of the **MessageManager** interface to send the update to the database. Once everything is done the client will be redirected back to the board page.

2.2.5 createBoard

This last servlet is invoked by the form inside the `index.jsp` page and it send a message to the database containing the name of the board that has been created by the user. The message is sent through the **MessageManager** function called **sendCreateBoard**, then the client is redirected to the page of the new create board.

2.3 Message Manager module (Jinterface)

This class manages the communications with the Erlang nodes using the Jinterface library. We decided to use a new version of this

library that has not been directly developed from Ericson, who is not updating the official one since 2016. The reason behind this is choice is that we have found some bugs with our Java version. These bugs have been largely notified to Ericson but no updated has been released, that is why the community decided to update the library by themselves.

We will have a look to the main functions that make the communication with the Erlang nodes possible.

- `OtpErlangObject`
`sendAndWaitForResponse(OtpErlangObject obj)`

This is a utility function used to send a generic message to the primary Erlang node and then wait for a response. The important thing to note is that if the webserver receives no response then the primary will be deemed as not reachable. After that RabbitMQ will be used in order to retrieve the PID of the new primary server.

- `ArrayList<String> loadBoards()`

This function allows the web server to request a list of all boards. In this function we build a message with the following format: `<load_boards, {}, primary, self>`

Notice that `load_boards` and `primary` are atoms. With `{}` we mean an empty tuple; this one is actually used in all messages to pass parameters but here we have no one to specify. We are still using it because the receiving Erlang node will append a timestamp to this tuple. Once the message is sent the webserver calls the wait for a response using the `sendAndWait` function. When a response message is received it is parsed to retrieve all the boards name, then they are returned inside an `ArrayList` object.

- `UpdatePackage loadTasks(String board)`

This function retrieves all tasks related to a specified board. Tasks are sent in an array format from the primary server, then they are re-arranged into stages list, finally everything is returned as an `UpdatePackage` object (which contains 4 lists, one for each stage). The message sent to the server has the following format:

`<load_tasks, {"BoardTitle"}, primary, self>`

Then the webserver waits for a response message with this format:

```
<[{"BoardTitle","Description",ExpirationDate",stage_id,"Title","Creator","Type"},{...},{...}]>
```

Things to note here is stage is indicated under the form of an integer called `stage_id`, which starts from 0 (that corresponds to Backlog stage) and goes up to 3 (that corresponds to Done stage).

- `sendCreateBoard(String board)`

This method is used when a new board is created, actually it sends a new message to the primary with the new board name and waits for an ACK from it. The ACK is very important since if a primary server goes down when this message is sent, database will be in an inconsistent state until a new primary goes up. Another case could be when a server receives the message but for some reasons (like a duplicated board name) gets an error when creating the board, if this happens the webserver will not receive and ACK and the board will not be displayed to the user.

The sending message format is the following:

```
<create_board,{"BoardTitle"},primary,self>
```

Receiving ACK format instead has this:

```
<ack_create_board,"BoardTitle">
```

An exception is raised in case of errors with the ACK or with an invalid board title.

- `void sendMoveTask(String board, String taskTitle, int toStage)`

If the user wants to move a task from a stage to another the webserver has to call this method in order to notify the update to the primary server. Also in this case the webserver will wait for an ACK, if this is not received the method will raise an exception.

The format of the sending message is the following:

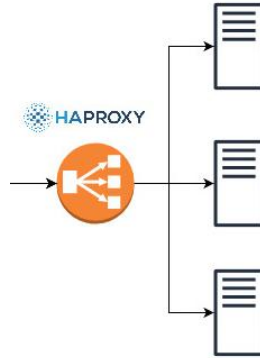
```
<update_task,{"BoardTitle"},"TaskTitle",new_stage_id}.primary,self>
```

The format of the receiving ACK instead is:

```
<ack_update_task,{"BoardTitle"},"TaskTitle",new_stage_id}>
```

2.4 HAProxy

During the development of this project, we decided also to distribute the web server modules in order to improve the overall performances and reliability of the environment. We used HAProxy, which is one of the most famous open-source solutions for workload distribution.



Setting up HAProxy for load balancing requires you to do is tell HAProxy what kind of connections it should be listening for and where the connections should be relayed to. We set up 3 different instances of web servers and we registered them inside the HAProxy configuration file `/etc/haproxy/haproxy.cfg`. Then we used as load balancing algorithm the Round Robin one, so each server is used in turns according to their weights. This is the fairest algorithm between the one that HAProxy offers, and it is also dynamic, which allows server weights to be adjusted on the fly.

This is a pretty basic setup, although this allowed us to increase the performance and the availability of the web servers, the only issue here could be HAProxy itself being a single point of failure. This issue can be fixed with floating IP between multiple load balancers but since we are still working with a pretty limited number of web servers this solution is not worth.

3 Erlang Nodes

3.1 Data consistency

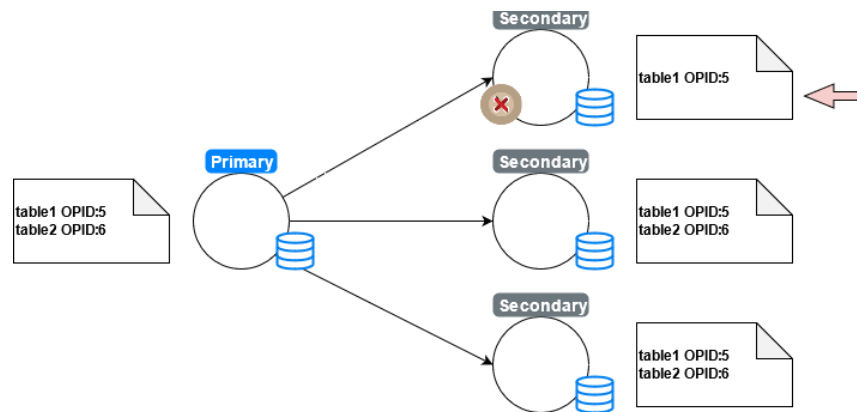
The chosen architecture is a primary-secondary architecture.

The primary sends back to a client requesting the data only after having received the ack from all the secondaries.

The read operation are performed to the primary to guarantee the consistency of the data.

3.2 Host failure

Secondary –



A primary broadcast all the write operations to the secondaries and wait back for an ack.

If an ack is not received from one or more secondary after 2 seconds:

The primary:

- Registers the host name and the operation_id the host did not responded back in the host_recovery table.
- Delete the host from his list of secondary hosts available in the cluster.
- Broadcast to all the secondaries the information's about the host which failed.

Secondaries:

- Registers the host name and the operation_id the host did not responded back in the host_recovery table.

- Delete the host from his list of secondary hosts in the cluster.

Primary –

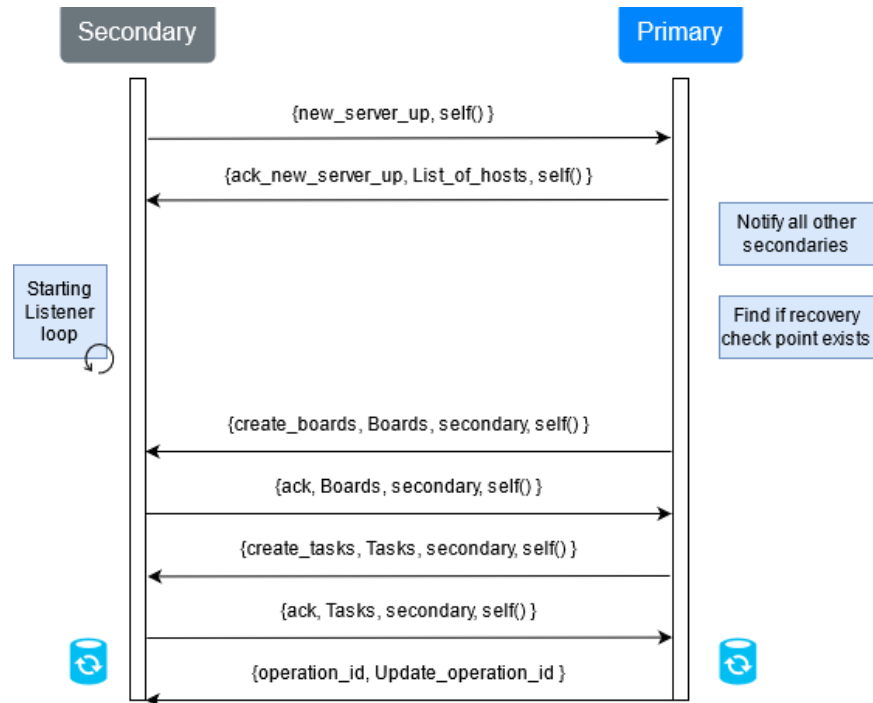
Since the secondaries are all up to date, any host in the network at the moment of failure can become a primary.

- Primary fails while no message was being processed.
- Primary fails sending data without being able to perform the query. Since client will not receive response will resend the request after checking RabbitMQ for the new primary.

3.3 New server up

When a new server is up, it sends a message to the primary.

- The primary notifies all the other hosts in order to add it as a new host on the cluster.
- Check if the host is a new host or a host which failed.
- Update its database from the point of failure or from scratch.
- Send the current operation_id so in case the primary fails without sending any message the secondary will have the operation_id updated.



In order to avoid sending many data a recovery policy has been implemented:

- Each performed insert or update is identified by an incremental `operation_id`;
- This `operation_id` is managed by the primary and send to the secondaries.
- When a secondary fails, the `operation_id` at which did not respond is stored in the database;
- When a failed secondary is back up, the primary will send only the data from the point of failure in order to avoid sending all the database back

4 RabbitMQ

Inside the system many Web Servers are employed to manage requests from clients, and the number of Web Servers can change dynamically.

Whenever a client makes a request to a webserver, e.g. update a task of a particular board, this change must be reflected on the application's view of the other clients who are interested in the same board. If more than one client is working on the same board, and one of them makes a change on one of its tasks, then the other clients must be informed too. For this purpose, we could have used two different approaches:

The primary informs all web servers of this change through a broadcast message, bringing a huge burden both on the network and the Primary Erlang Server

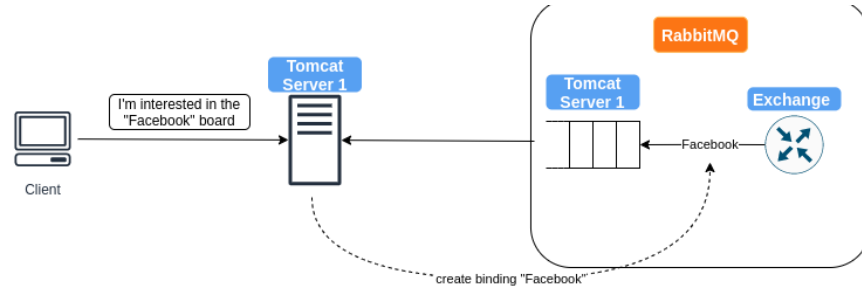
Use a Queueing System which can alleviate the load on the primary server, in fact: with this solution only one message is sent by the primary server to inform all web servers.

4.1 Architecture Overview

Whenever a new web server is started a new queue is created inside RabbitMQ. This queue is personal to that specific webserver and will contain all update messages directed to it.

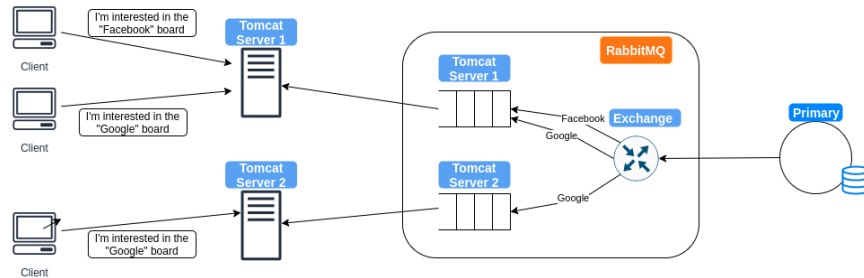


When a new client opens a board, meaning it is interested in receiving updates for that specific board, a new binding is created. The purpose of this binding is to inform RabbitMQ that, from now on, every update received from the Primary Erlang Server regarding that specific board must be forwarded (also) to that specific Web Server.



In a more complex scenario, we can observe how this architecture works. Suppose an update to the board “Google” is coming from the Primary Erlang Server:

1. The message is received from RabbitMQ’s exchange which forward the update to a set queue based on the binding previously created.
2. In this case, the exchange redirects the message to both queues since a “Google” binding is present for both.
3. Finally, the update will be received from both Web Servers which, in return, will notify the clients of the changes.
- 4.



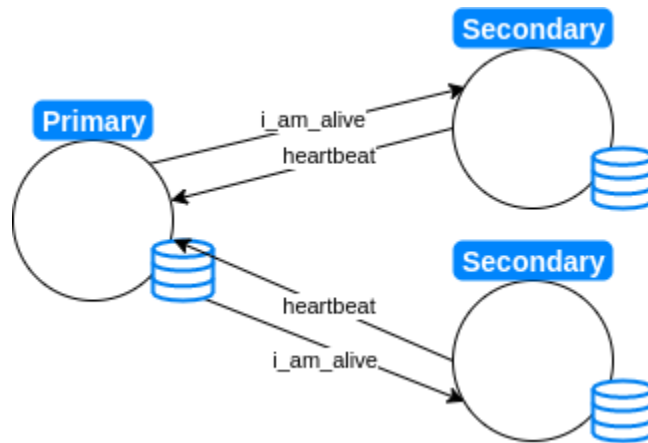
4.2 Primary Failure Detection

To provide fault tolerance to the system, Secondary Erlang Server must be aware of the state of the Primary and, in case of failure, start a new primary election.

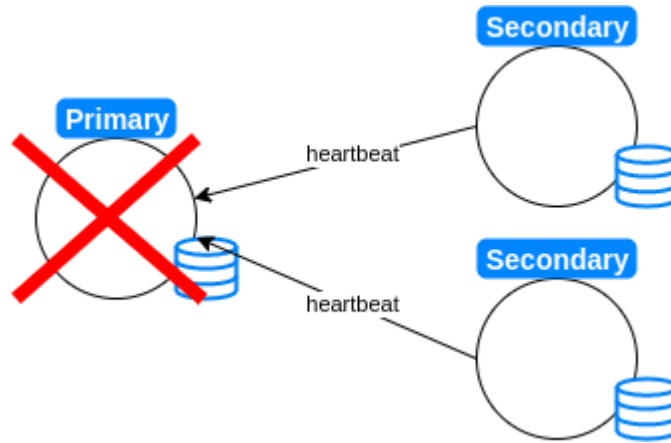
The algorithm used to identify a failure of the Primary Erlang Server is quite simple:

1. Every 2 seconds, each secondary send an ‘heartbeat’ message to the Primary

2. If the Primary is up, it responds with an 'i_am_alive' message and the algorithm ends



3. Otherwise, if the Primary is down, no 'i_am_alive' message is received from the secondary servers, which in return will start an instance of the Bully Algorithm (election of a new primary)



4.3 Bully Algorithm

The algorithm assumes that:

- The system is synchronous
- Erlang servers may fail at any time, including during execution of the algorithm
- A process fails by stopping and returns from failure by restarting

- There is a failure detector which detects failed processes (see “Primary Failure Detection”)
- Message delivery between processes is reliable
- A list of erlang process PIDs participating in the cluster is maintained by every node. This list is the same for each server and it’s used during the election to identify which server has higher priority (PID closer to the list’s head)

4.3.1 Algorithm

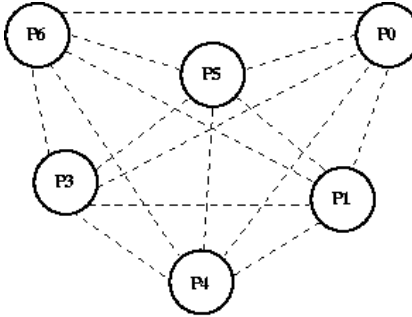
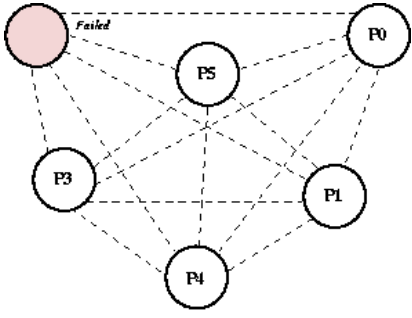
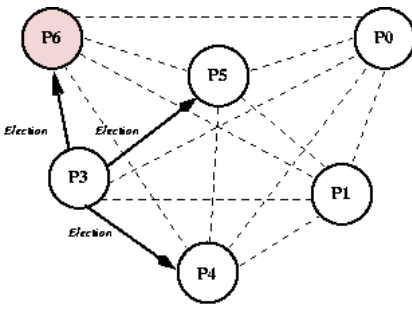
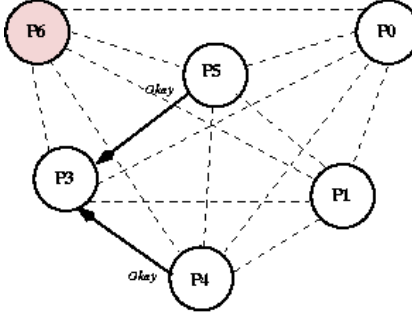
The algorithm uses the following message types:

- Election Message: Sent to announce election.
- Answer (Alive) Message: Responds to the Election message.
- Victory Message: Sent by winner of the election to announce victory.

When a process P recovers from failure, or the failure detector indicates that the current primary has failed, P performs the following actions:

1. If P has the highest process ID, it sends a Victory message to all other processes and becomes the new Primary. Otherwise, P broadcasts an Election message to all other processes with higher process IDs than itself.
2. If P receives no Answer after sending an Election message, then it broadcasts a Victory message to all other processes and becomes the Primary.
3. If P receives an Answer from a process with a higher ID, it sends no further messages for this election and waits for a Victory message. (If there is no Victory message after a period of time, it restarts the process at the beginning.)
4. If P receives an Election message from another process with a lower ID it sends an Answer message back and starts the election process at the beginning, by sending an Election message to higher-numbered processes.
5. If P receives a Victory message, it treats the sender as the Primary.

4.3.2 Example

<p>We start with 6 processes, all directly connected with each other.</p> <p>Process 6 is the leader, as it has the highest number.</p>	 <p>Bully Algorithm: Step 0</p>
<p>Process 6 fails.</p>	 <p>Bully Algorithm: Step 1</p>
<p>Process 3 notices that Process 6 does not respond so it starts an election, notifying those processes with ids greater than 3.</p>	 <p>Bully Algorithm: Step 2</p>
<p>Both Process 4 and Process 5 respond, telling Process 3 that they'll take over from here.</p>	 <p>Bully Algorithm: Step 3</p>

