Chapter 1

System Architecture and Behaviours

1.1 Introduction

In the ?? and ??, the logical layering and the responsibilities of the components were introduced but how they fit together and work in harmony is still a mystery. In this chapter, I will put the pieces of the puzzle together by discussing of the actual architecture of the system and its behaviours with increasing levels of detail.

1.2 Top-level architecture

The top-level architecture is a simplified architecture where many low-level components are being grouped together and treated as a singular composite component. It is shown in the figure 1.1. The following components in the top-level architecture are composite component:

- Remote Sensing (See subsection 1.4.1)
- Backend (See subsection 1.4.2)
- Frontend (See subsection 1.4.3)
- Real-time (See subsection 1.4.4)
- Governor (See subsection 1.3.3)

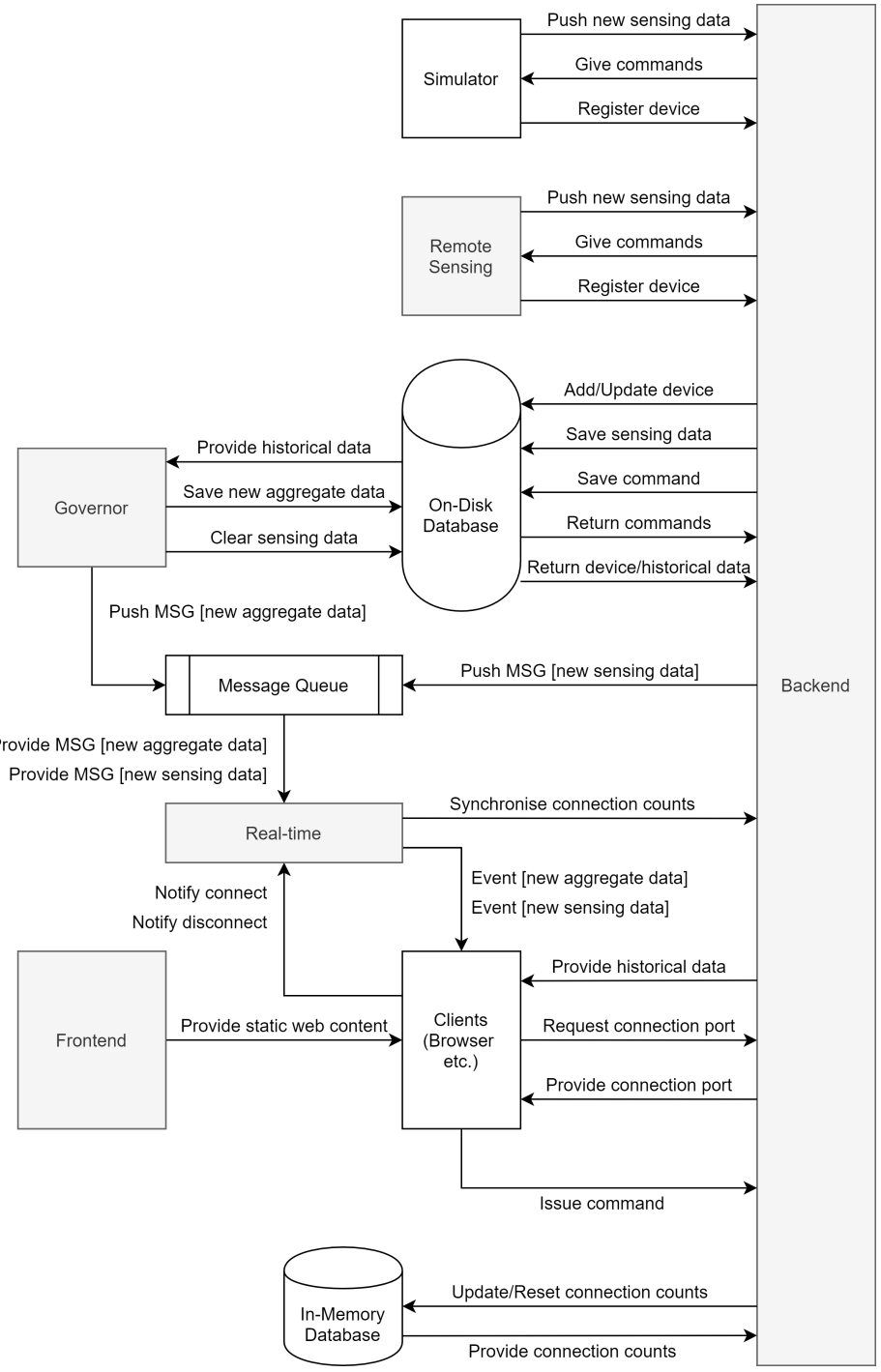

Figure 1.1: Top-level architecture of the proposed system.

1.3 Top-level behaviours

This section provides a general description of how the the proposed system works together with many low-level details abstracted.

1.3.1 Initialisation

The initialisation process is referring to a device in the remote sensing composite component such as a microcontroller attempting to connect with the server before it can send data and receive commands. When a microcontroller is initialising, it sends a register device request to the backend and then wait for the reply from the backend. Once the request is received by the backend, it tries to find a matching device in the database. If found, then it updates the device status to online. Otherwise, it generates a new device ID, creates a representation of the new device in the database, and set the device status to online. Finally, the backend notifies the microcontroller by returning a response to the initial device registration request, and the intialisation sequence is complete. The figure 1.2 shows the flow of the initialisation process and the relevant function calls shown in the top-level architecture.

The initialisation process achieves two goals, the first goal is letting the backend knows the device is online and ready to transmit data, and the second goal is getting a unique ID so that it can be identified during the data transmission phase. Once the initialisation process is finished, the microcontroller is ready to transmit data.

1.3.2 Data recording and device controlling

The data recording process aims to record data from the sensors and store them into the database. The process begins at the microcontroller where it reads the sensing data from the sensors and then sends it to the backend using the *push new sensing data* request. Once the backend received the sensing data, it begins executing two sequences of instructions simultaneously. The first sequence is used for data recording, where the data is added to the database and then sends a message to clients notifying a new sensing data is available. The second sequence is used for device controlling, where it retrieves a list of commands cumulated in the database, and sends them to the microcontroller through the response to the *push new sensing data* request. As soon as the microcontroller received the commands, it would perform each command in order and wait for 1 second before reading and sending data again. The process is shown in the figure 1.3.

The reason for data recording and device controlling happening at the same time is improving communication efficiency and reducing server load. The micro-

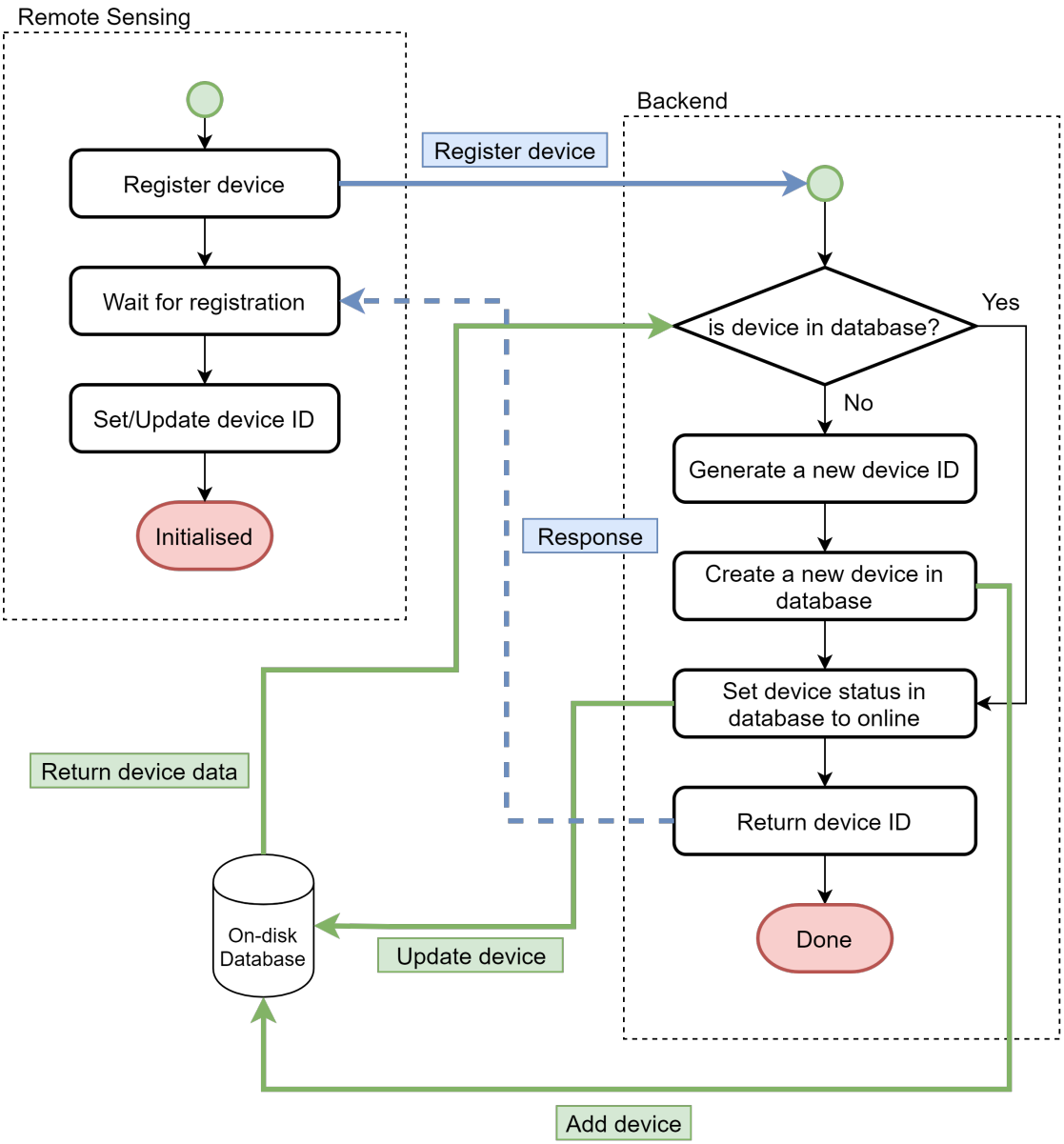

Figure 1.2: The initialisation process.

controller doesn't need to send two different requests to push data and retrieve commands, which reduces the communication time by half and reduces the number of access to the backend by half.

Executing two seqeunces of instructions simultaneously has the benefit of reducing the response time of a request. The detailed explanation is provided in subsection 1.4.2.

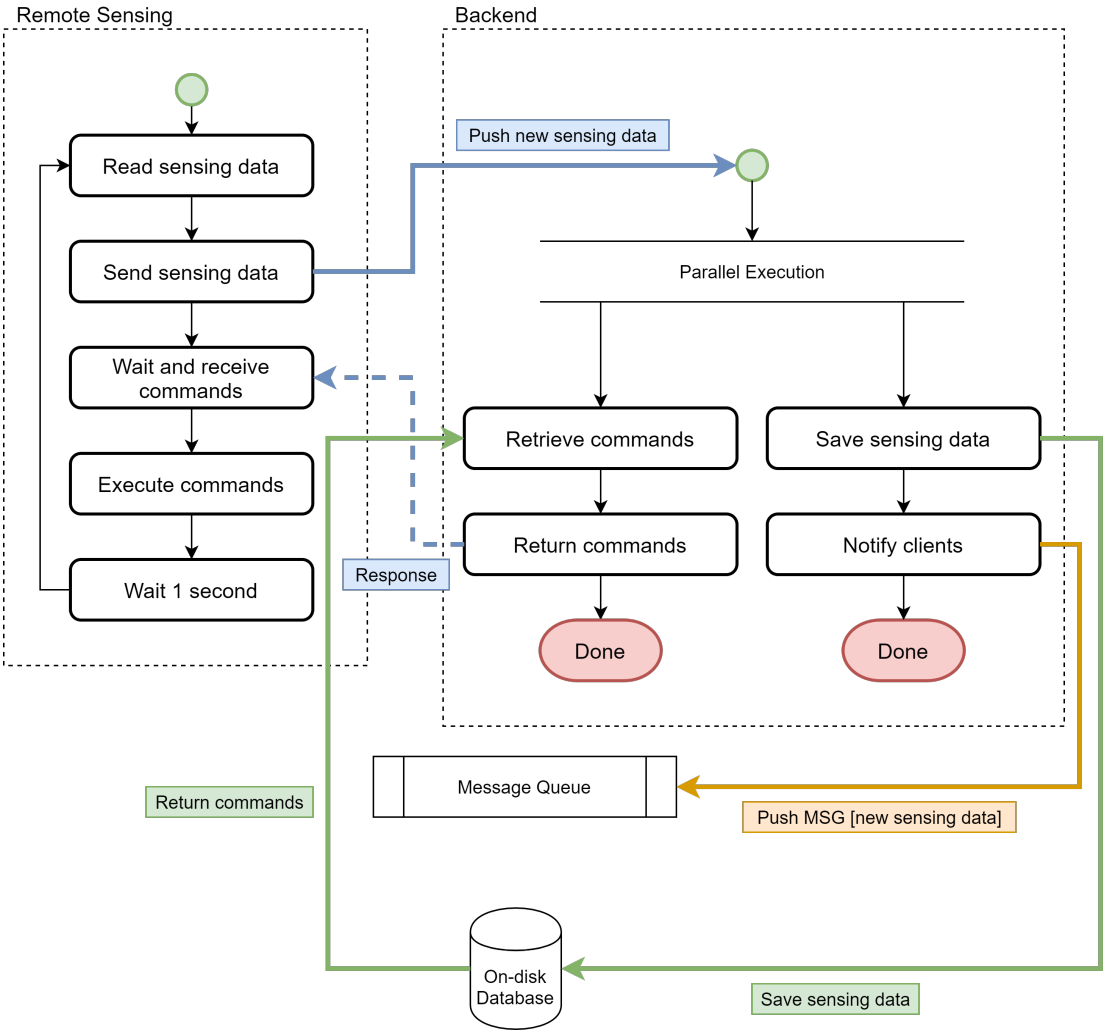

Figure 1.3: The data recording and device controlling process.

1.3.3 Governing

Governors are a series of background tasks that are executing periodically and making changes to the system. Currently, there is only one governor in the system, the data aggregation governor.

Every minute, the governor aggregates the sensing data[1] for every single device by averaging the sensing data received over the last minute, saving it to the database, and notifying clients about a new aggregate data is available. To prevent the database using too much disk space, it resets the sensing data after aggregation so the number of samples in the sensing data is kept below 60 samples. The

[1]The sensing data is also known as the real-time data since it is recorded every second.

governing process is shown in the figure 1.4.

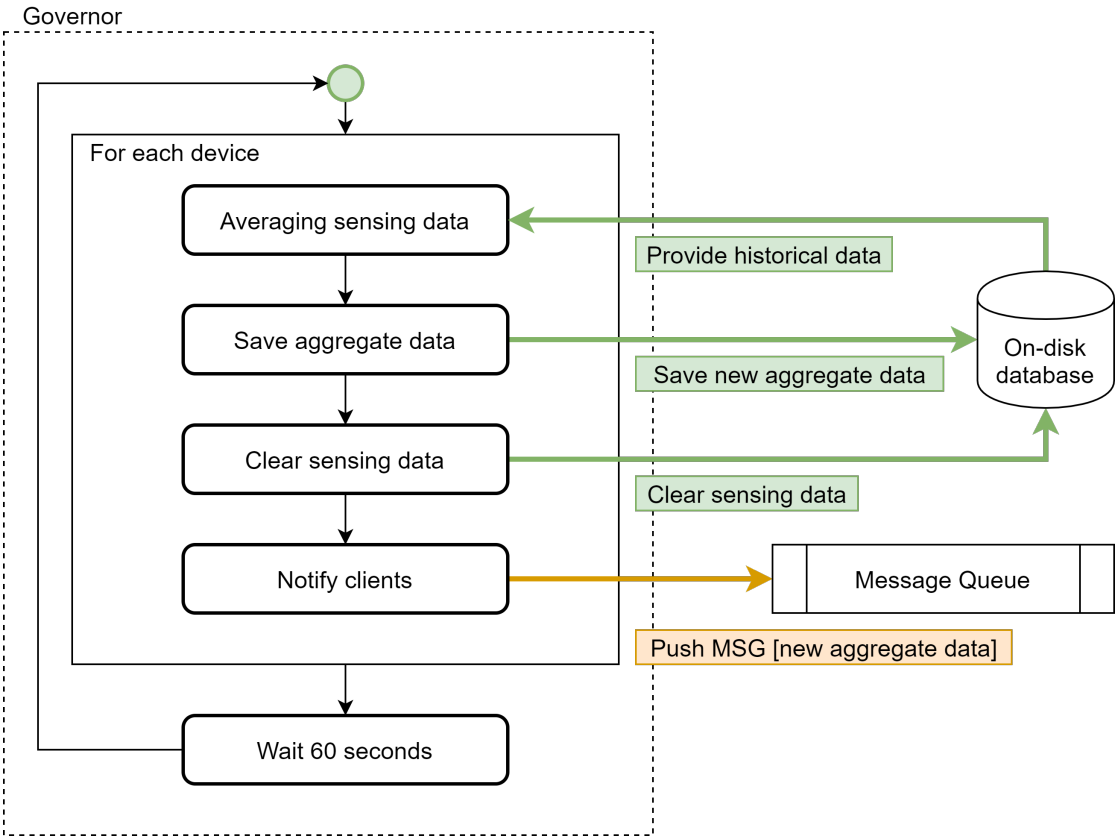

Figure 1.4: The governing process.

1.3.4 Context-aware load balancing

Load balancing is an act of distributing load to different servers evenly to improve the overall performance. This is because it prevents the situation where a single server is overwhelmed by the load while other servers are experiencing starvation[2]. It is context-aware in the proposed system because the load balancer keeping track of the number of active connections a server has and direct the load to the server with the least amount of active connections.

This is different to the conventional load balancers where it uses simple scheduling algorithms such as round-robin to dictate which server the load should be distributed to. The primary reason for using a custom scheduling logic is because

[2]Starvation in the context of computer system is a resource such as computation power that is being significantly underutilised.

the connections in the proposed system are persistent[3] while the connections in a typical system are one-off[4]. That means, if a server is unfortunate enough that all of its active connections have very long lifespan, while connections in other servers have very short lifespan. The simple scheduling algorithm would continue to distribute the load to the server that still has many active connections due to the algorithm only cares about the order of the distribution not the actual load the server has. By being context-aware, the load balancer only distribute the load to the server with the least amount of active connections and the scenario above is avoided.

Back to the proposed system, the context-aware load balancing process is shown in the figure 1.5. For a connecting client, it starts by querying the backend for a port it should connect to[5]. Then, the backend retrieves the connection count for each port from the in-memory database, find the one with the least amount of active connections, and return the port to the client. Once the client knows the port, it connects to the server at the port (the real-time composite component) and notify the server that it has connected. The server then realise it is a connect event and increment its internal connection count which keeps track of its own number of active connections. Finally, the server tells the backend that its active connection count has increased so that the backend is aware of the change and update the connection count associated with the port in the in-memory database.

For a disconnecting client, which is caused by closing the web application or disconnected due to netowork issues. The disconnection would be detected by the server automatically and decrease the internal connection count. Like before, the server tells the backend that a connection has disconnected and the count is decreased. The backend reacts to it by also decreasing its connection count associated with the port that are stored in the in-memory database.

Notice that both connecting and disconnecting would trigger a connection count update in the backend, and this allows the backend to keep track of the individual server load and distribute the load properly.

The reason for using an in-memory database as oppose to storing the connection count in the server instance is because there are more than one backend instances in the proposed system. If all of them store the connection count in their own instance, then the connection count would not be updated correctly as the connection count synchronisation request is randomly distributed to any of the instances. As a result, each backend instance has a different connection count. Therefore, they need to share the connection count data using the in-memory database so all instances are accessing and updating the same connection count.

[3]The connection may last for a long time, and making many queries while it is connected.

[4]The connection only query the system once and only once.

[5]A server usually sits behind a port in the computer system. Therefore, querying for a

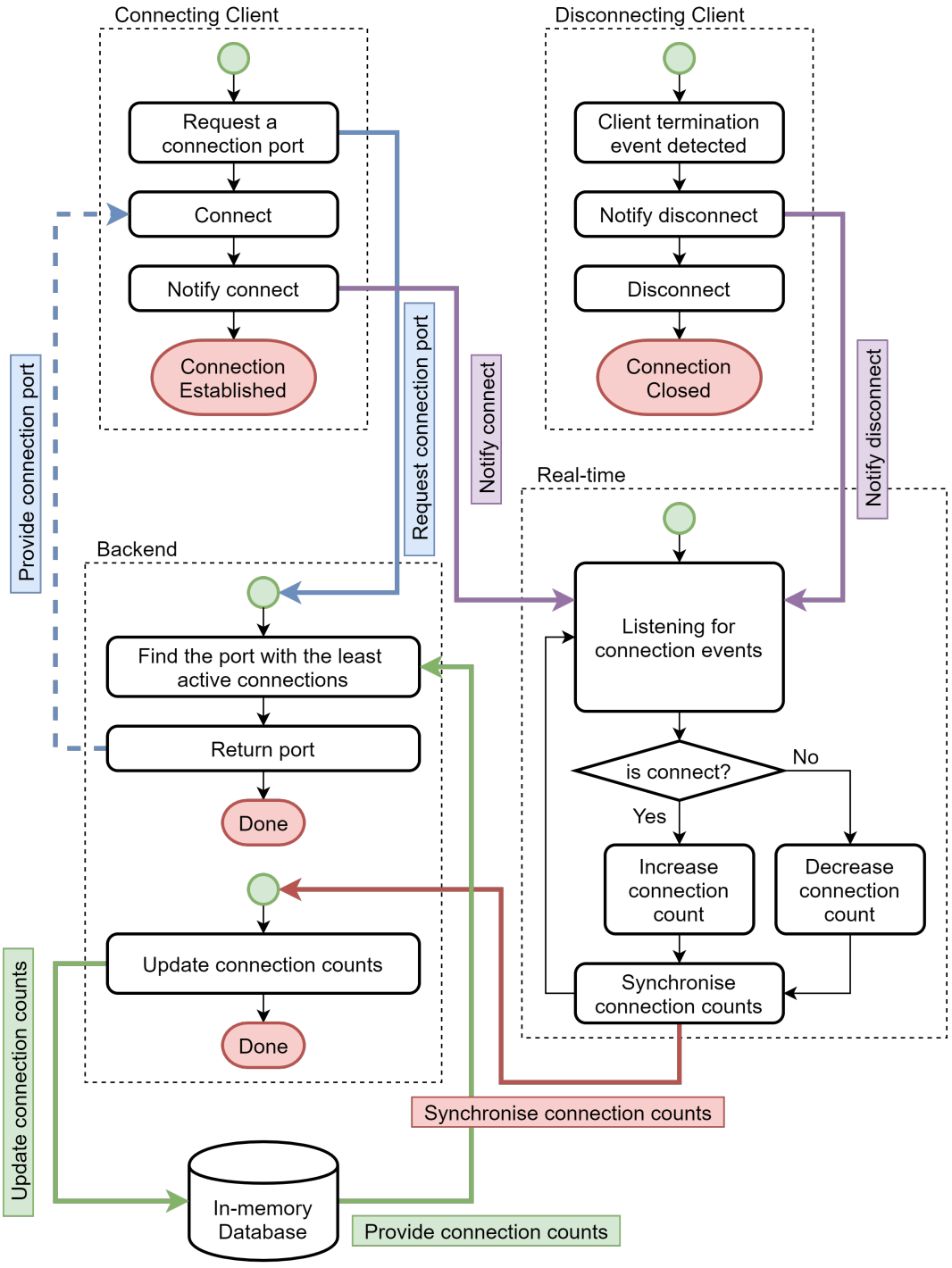

Figure 1.5: The context-aware load balancing process.

connection port is the same as querying for a particular server it should connect to.

1.3.5 Data monitoring

The data monitoring process is about showing the data on the client and the process is shown in the figure 1.6. Once the client is connected to a real-time server, the data monitoring process can begins. The first step is preloading the historical data from the backend which allows the user to view the previously recorded data for upto 4 hours immediatly after the client establishing the connection. During data preloading, the backend is fetching the data from the on-disk database and provide it to the client in the required format. Once the data preloading is done, the client starts listening for incoming data from the real-time server. Recall the new aggregate data is generated by the governor and the new sensing data is generated by the microcontroller, both of them produce a message into the message queue signifying a new piece of data is available. The consumer of both messages is the real-time server where it listens for the new data messages from the message queue and broadcast the new data to the relevant clients. The client takes the new data and add it to the visualisation.

The benefit of using the real-time server is it significantly reduces the amount of backend access. Without the real-time server, the client have to periodically querying the backend regardless if there are new data or not. This is because there is no way to tell if a new piece of data is available at any given time. Assuming the querying rate is one query per second and there is only one client querying the backend. The periodic querying method would generate 60 queries per minute. By contrast, the real-time method only generates one and only one query, which is used for preloading the data. The load for the real-time method is shifted from backend to the real-time server and the number of messages from the real-time server is the number of times the microcontroller records the data. That means the worst case scenario is generating 60 data update messages per minute [6] to the client and the best case scenario is generating no data update as the microcontroller is not recording data. It is also worth noting that the cost of a query and data update is very different. A query involves accessing the on-disk database whereas the data update doesn't. Since accessing on-disk database is very slow by contrast, 60 real-time updates is more efficient than 60 queries.

1.3.6 Issuing command

The process of issuing a command involves the user issuing a command through the user interface and queuing the command in the database for the microcontroller to retrieve and execute. The process is shown in the figure 1.7. The process begins when the client sends the command to the backend, and then the backend enqueue the command inside the on-disk database. The order of the commands stored in

[6] The microcontroller records one sensing data every second.

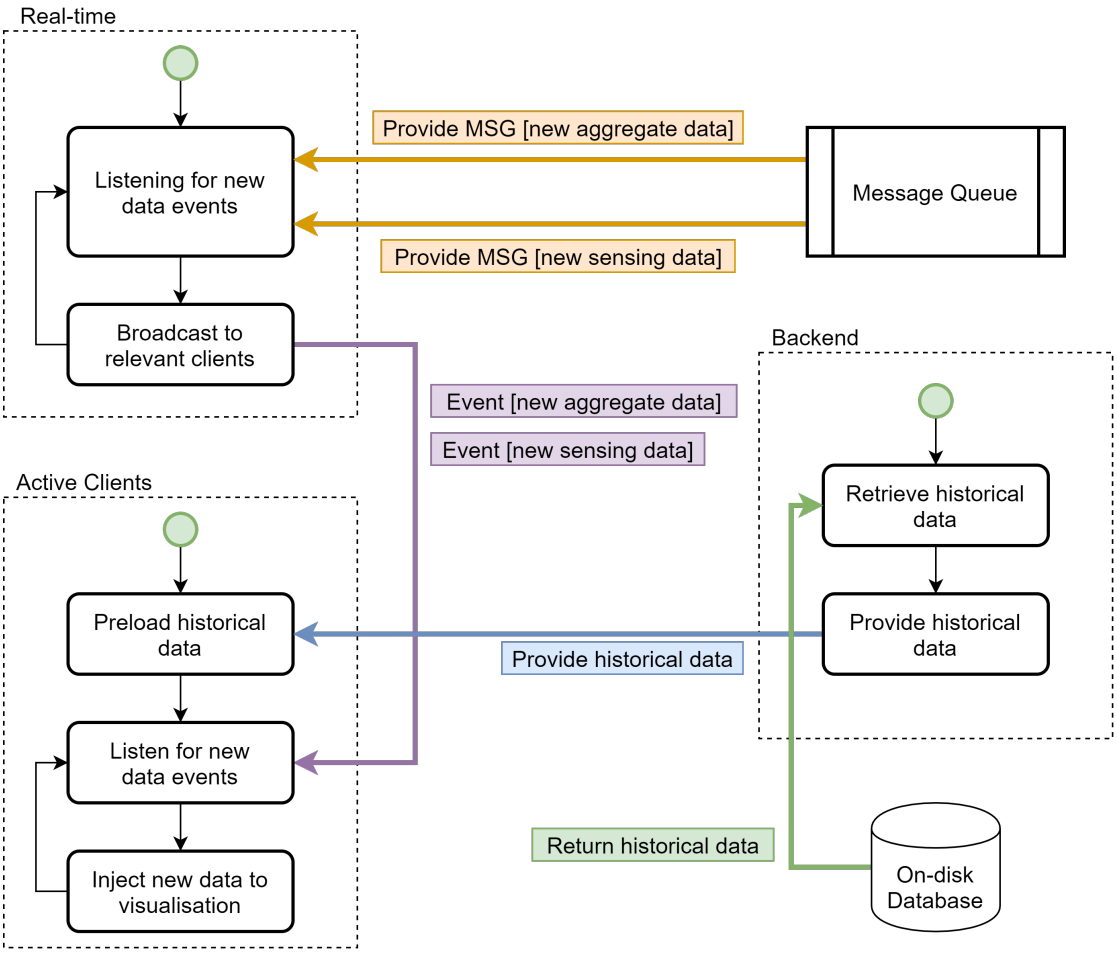

Figure 1.6: The data monitoring process.

the database must be maintained to ensure the correctness of the result after the microcontroller executes a series of commands.

1.4 Low-level architecture

In this section, the composite components shown in the top-level architecture are described in detail.

1.4.1 Remote sensing and simmulation

The figure 1.8 shows the remote sensing composite component is made up of the following low-level components in the experimental setup:

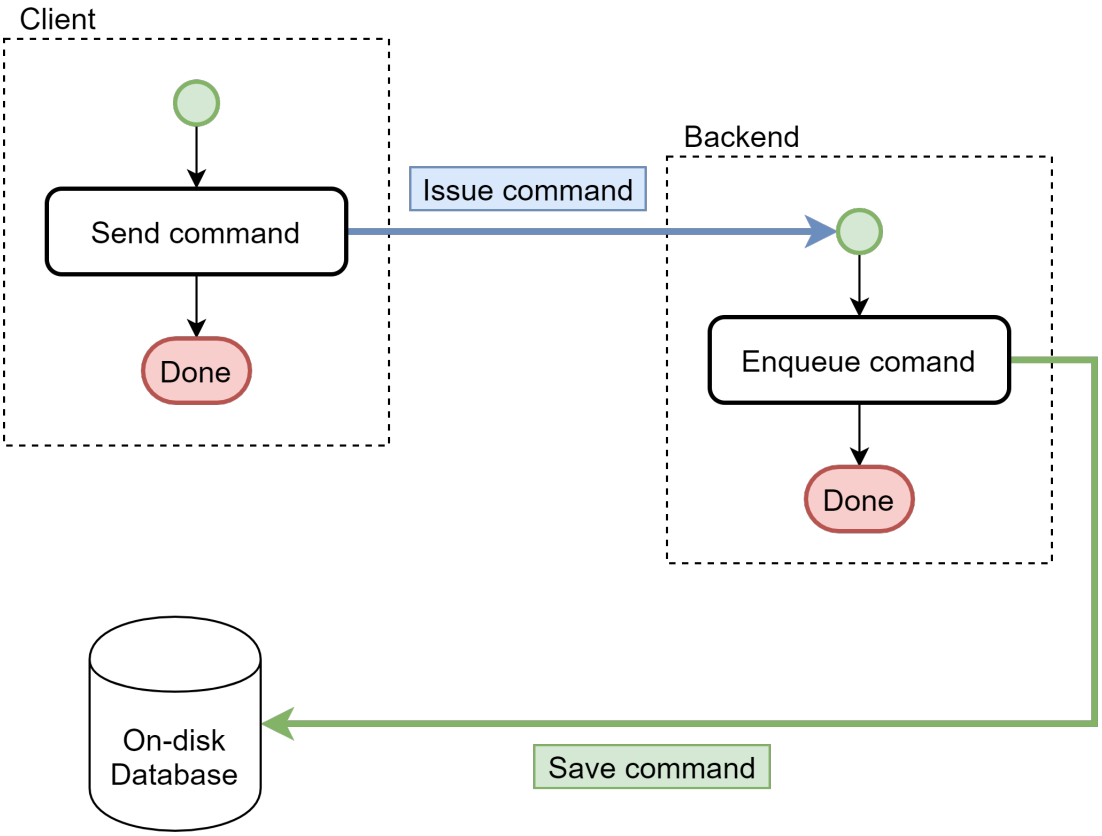

Figure 1.7: The process of issuing a command.

- Solar Panel
- Sensor
- Converter
- Battery
- Microcontroller
- LTE Modem
- Antenna

In the experimental setup, the solar panel generates electricity which is measured by the input sensor and then fed into the converter. After converting the electricity to the desired voltage, it is measured by the output sensor and stored

in the battery. Then, the microcontroller reads the analog output from both sensors, computes the current and voltage readings, and pushes new sensing data to the backend through the LTE Modem using HTTP protocols. Additionally, if the commands are received, the microcontroller would send electrical signals to the converter to alter its behaviour.

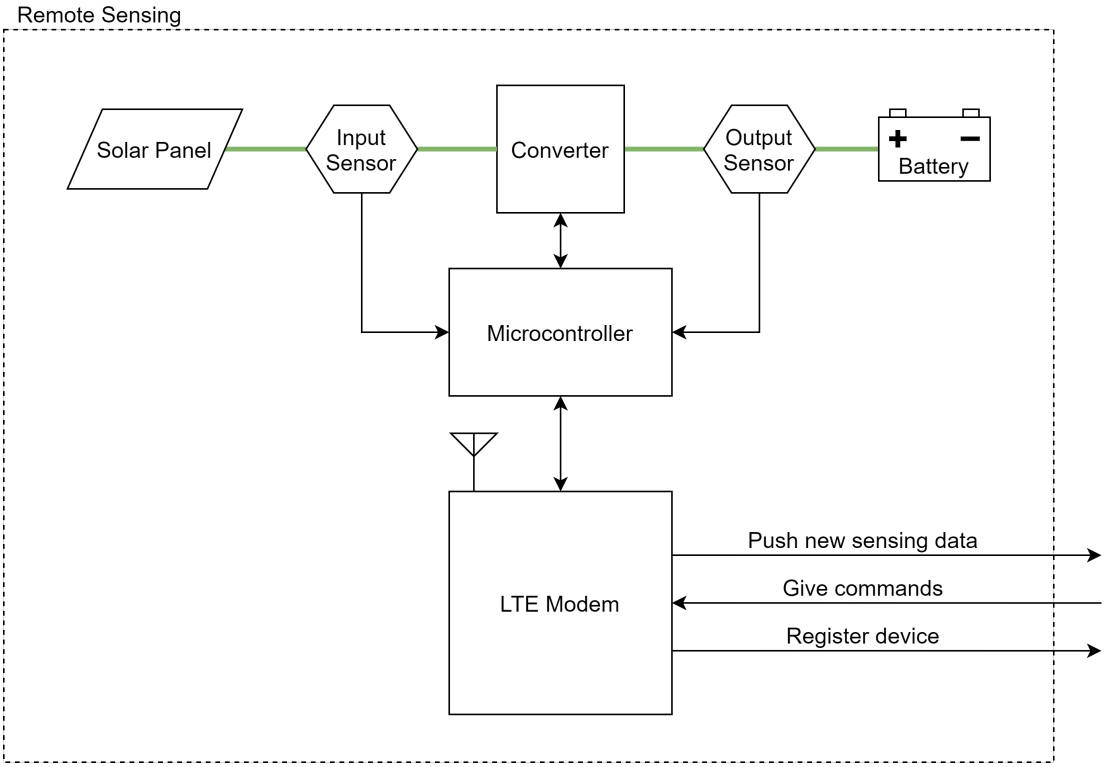

Figure 1.8: The architecture of the remote sensing composite component.

The low-level components can be varried greatly depending on the use cases, as long as it confronts to the behaviours that the proposed system expects:

- Registering with the backend.
  - Providing its device ID to the backend on startup, or request a device ID if it doesn't have one.
  - Remembering its assigned device ID given by the backend.
- Sending sensing data to the backend.
  - Current from the power generation device. (Input current)
  - Voltage from the power generation device. (Input voltage)

  - Current from the power converter. (Output current)
  - Voltage from the power converter. (Output voltage)

- Executing commands.
- Communicating with the backend over HTTP protocols.

A simulator was developed which confronts to the expected behaviours of the proposed system to generate realistic server load for benchmarking. The simulator has the capability to spawn hundreds or thousands of computer processes and each simulating a remote sensing component that is recording data and pushing them to the backend.

1.4.2 Backend

The backend composite component is shown in the figure 1.9 and consisting of the following components:

- Load Balancer
- Backend Instances
- Job Queue
- Async Worker

All requests to the backend must go through the load balancer, which distributes the requests evenly to the two backend instances that are actually handling the requests and generating the responses. Among those requests, some of them are time insensitive[7]. This type of tasks are added to the job queue which allows them to be processed later. Finally, the jobs in the job queue are taken and executed by the async worker in a first in first out manner.

To understand how this processing mechanism allows more requests to be processed, we need to understand a common metric to measure the performance of a server, the response time. The response time is the time from sending the request to receiving the response. Therefore, the response time can be broken into transmission time and processing time. The transmission time is the time it takes for a request to travel from the sender to the backend and the time it takes for a response to travel from the backend to the sender. The processing time is the time it takes for a request to be processed at the backend. Furthermore, the tasks are either time sensitive[8] or insensitive, we can break the processing time into

[7]Tasks that we don't have to wait for the result to be availabe before we proceed.
[8]Tasks that we must wait for the result to be availabe before we proceed.

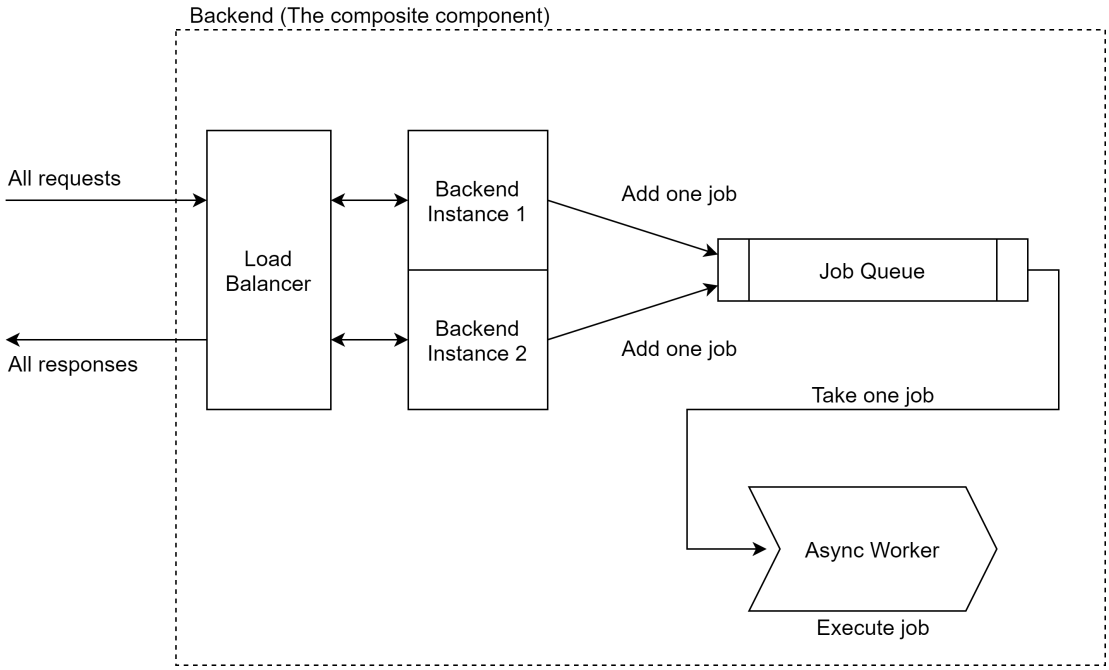

Figure 1.9: The architecture of the backend composite component.

the time it takes to execute time sensitive and insensitive tasks. If the average response time of a request can be reduced, then the backend can process more requests given the same amount of time.

The figure 1.10 illustrates the async worker accelarates the response time of a request, which allows the backend to process more requests. Assuming the transmission time, the processing time of time sensitive tasks, and the processing time of time insensitive tasks are identical with and without async worker. Without the async worker, tasks must be executed one after another. Therefore the processing time includes the time to execute time sensitive and insensitive tasks. With the async worker, we can add the time insensitive tasks into the job queue, which allows them to be executed at a later time. This is fine because the result of the time insensitive tasks are not important to processing the requests. Therefore, the processing time includes the time to execute time sensitive tasks and the time to enqueue a task to a job queue. There is no need to wait for the time insensitive tasks to finish and the time to enqueue a task is usually much faster, which allows the request to be processed faster, reduces the response time, and improving the performance of the backend.

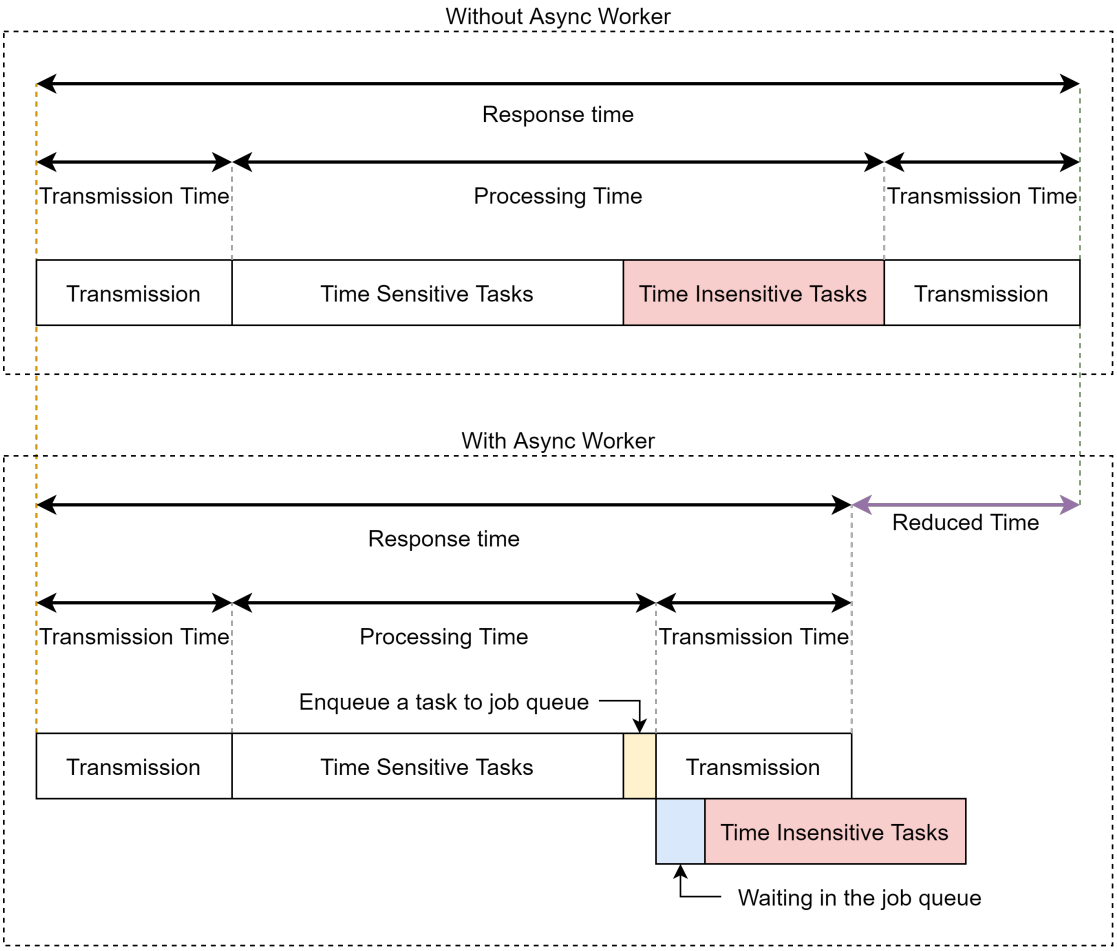

Figure 1.10: The comparison between using async worker and not.

1.4.3 Frontend

The frontend composite component is shown in the figure 1.11 and consisting of the following components:

- Load Balancer
- Static Web Content

Similar to the backend, all requests must go through the load balancer, and each thread in the load balancer is responsible for retrieving the static web content[9] and provide it to the client.

[9]Parts of the website that doesn't change.

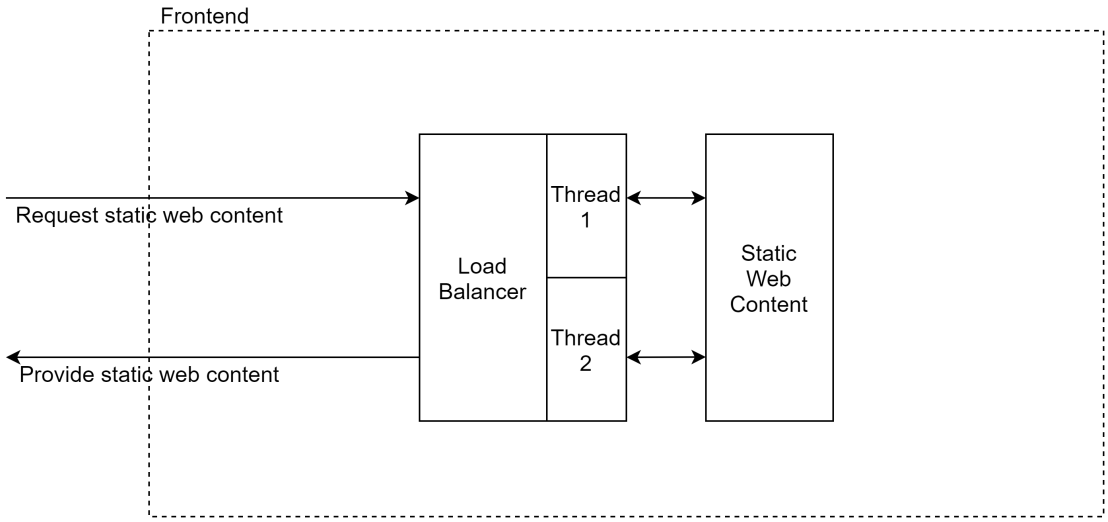

Figure 1.11: The architecture of the frontend composite component.

1.4.4 Real-time

The real-time composite component is shown in the figure 1.12 and consisting of the following components:

- Data Distributor
- Socket Instances

The data distributor receives data from external sources and distribute (emit) the data to the socket instances. Then, the socket instances send the data to their connected clients in the form of events.

To make it more efficient, the data distributor distributes the data to the socket instance if and only if the socket instance needs it. This is done by keeping track of what data that each socket instance needs. When a client is connected to the socket instance, the client tells what data that it wants to receive. Then, the socket notifies the data distributor through a connection tracking synchronisation request that the client on the socket want to receive certain data when it is available. Therefore, the data distributor knows if the socket needs a certain piece of data and send it to the socket. If the client is disconnected, the socket would send another connection tracking synchronisation request to the data distributor and notifying this client no longer needs the data.

A simple scenario is illustrated below which shows how the data distributor reacts to different situations.

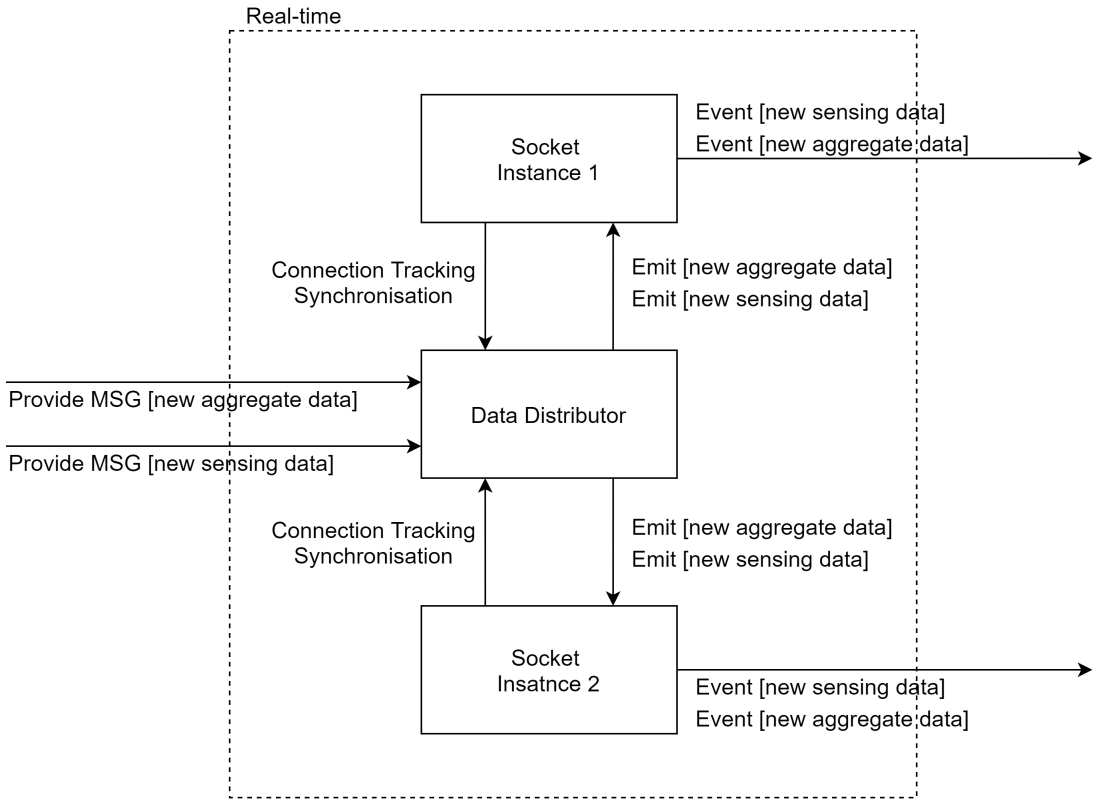

Figure 1.12: The architecture of the real-time composite component.