Distributed Systems

Project 4 – Fault Tolerance in a Distributed Environment

11th July 2014

David Simões, 60089
ECT

The Issue

- A **distributed system**, being made of multiple hardware components, is susceptible to a **wide array of system failures**. It is, therefore, a desired characteristic of the system the “ability to **tolerate and recover** from faults”.

- From the multiple types of failures, we will focus on the **system crash**, similar to when the power runs out, or the computer “blue screens”. This means that **higher-level threats** (such as a corrupted disk) **won't be tolerated**.

- How can the system recover from a crash? It will need to **resume it's state** from his previous one and perform his operations **like nothing has happened**.

State Machine

This will imply having a defined **state machine**. This state machine will be based upon the sequence of **instructions and interactions** that the given node will have with other peers.

Every state must be **well-defined**, so as to allow the proper **execution of instructions** it is assigned; the flow between the states must also be defined, to **remove any kind of ambiguity**.

When any node changes states, it will **save the current state** in the file-system so that, in case the computer crashes, it will be able to **read the state** it was in and resume his operations.

Detecting Crashes

- **Detecting another node has crashed:**
  - If the node hasn't connected, it's no problem.
  - If we can't connect to a node, it has crashed.
  - If we are connected and the node doesn't exchange information properly, it has crashed.
  - If we are connected and the node and exchanges information, it hasn't crashed.

- **Detecting self crash:**
  - If there is no previous state, we haven't crashed.
  - If there is a previous state, we have crashed and need to resume the previous state and continue with our execution.

Repeated Requests

We have resumed from a crash. **We may have completely interacted with someone and crashed before we could save a state**. It's impossible, however, to know whether we have interacted or not, in an efficient way.

This problem is fixed with a **Request ID**. Every interaction started has a unique ID for the given target node. Each new state will change the ID. **Repeated interactions will carry the same ID**, which can be detected by the receiver.

If the receiver receives duplicate IDs, then it will **re-send the last answer** it has sent to that node without actually performing operations (which will prevent a corruption of the distributed system's state).

State Saving

Saving the node state implies **saving all the internal variables, configuration values, and current thread execution position**. Not only that, but on a server node, multiple threads will require multiple states to be saved. This will be done at the start of each new state in the state machine.

Not an atomic operation! Saving the node's state requires creating a file and writing on it. The crash, however, may occur during or between these phases. Therefore, when recovering from a crash, the parsing must guarantee that the **state has been completely saved** to the file.

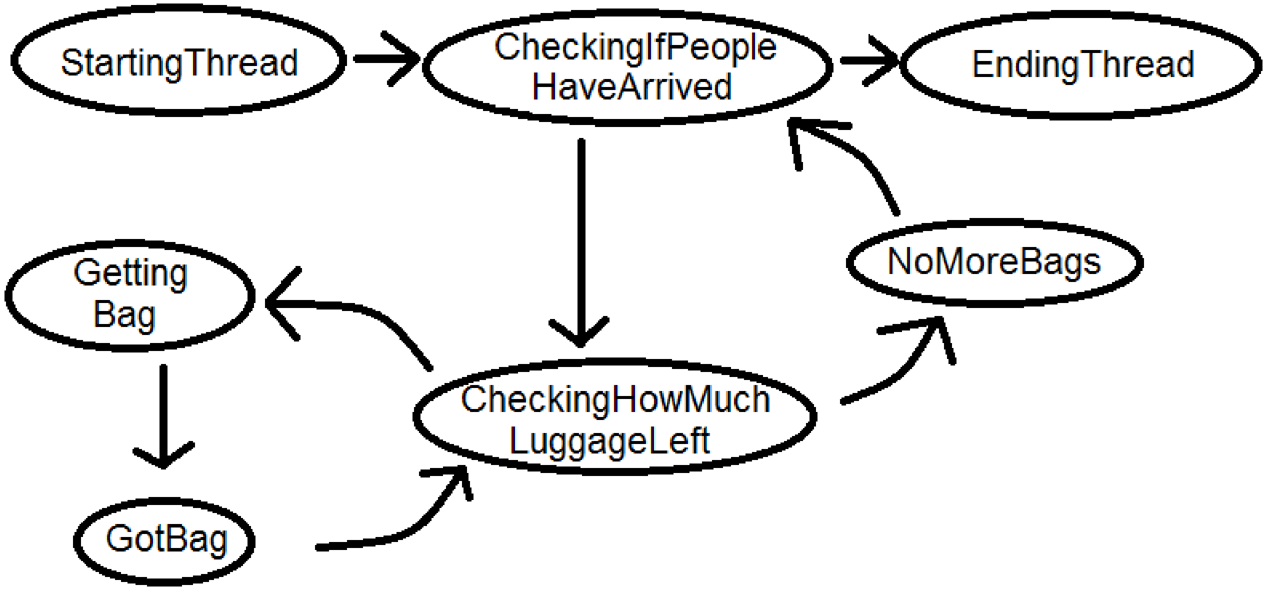
If a crash occurs while writing the new state, then the node must resume his previous state. Therefore, **the previous state must not be deleted**, but simply renamed (for example, adding “.old” to it's name) until it is no longer needed.

Adapting to our Airport

Fault tolerance has been implemented in a client and on a server: the **Porter** thread and the **TransferDeparture** monitor.

To implement states for the Porter, it was possible to create and save the state for every new method or function called, but it wouldn't be very efficient.

Instead, the **state** is saved **between every call to the same monitor**. This minimizes the amount of saves needed and also respects the need to keep unique Request IDs for each call.

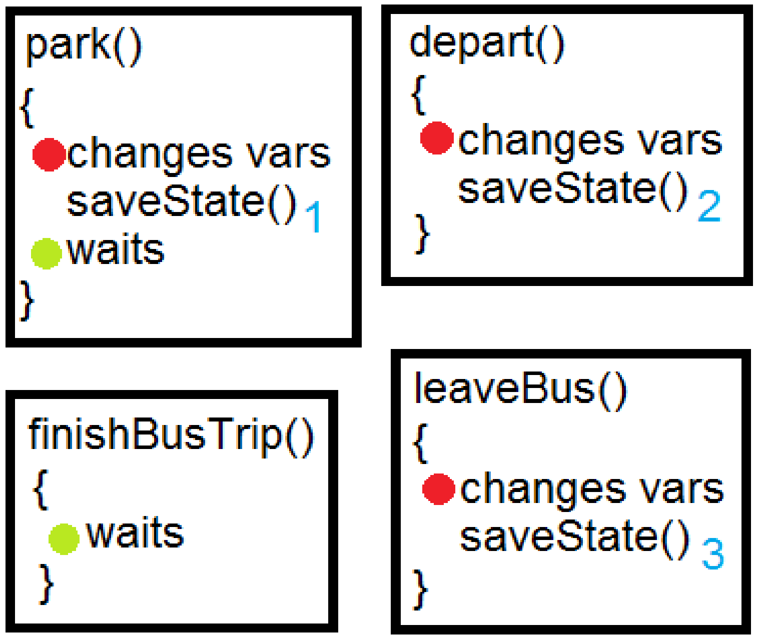


Adapting to our Airport

The states created for the monitor were done in a similar fashion. Keeping in mind that each thread will only call a single method, state diagrams were elaborated for each of those methods. However, these proved to be too complex for the given problem.

First of all, only methods that **changed the monitor's internal variables** needed a state save. Secondly, only when **leaving the critical region** does the state need to be saved. Combining these two conclusions will lead us to only 3 methods having states, and only a single state each.

Each thread state had to be identified by the ID of the caller and the Request ID, so as to not mix up the states. The monitor state (with the internal variables) is a different single file (otherwise, when resuming from a crash, the monitor wouldn't know which was the most recent state).



```
park()
{
 ● changes vars
   saveState() 1
 ● waits
}
```

```
depart()
{
 ● changes vars
   saveState() 2
}
```

```
finishBusTrip()
{
  ● waits
}
```

```
leaveBus()
{
 ● changes vars
   saveState() 3
}
```

File Format

PorterState(.old)	var;rID;stateID;
MonitorState(.old)	var;var;
MonitorThreadStateID;rID	stateID
MonitorThreadStateID;rID.temp	

For the Thread, the file format is simple and the way to make sure the file isn't corrupt is straightforward. For the monitor, however, the case is more complex. Since we need 2 files to save the monitor's current state, then we need a way to make sure that if the monitor crashes while only one of the files has been written, it will ignore that file.

A simple way of doing it is to create a temporary file which indicates whether the monitor crashed while saving a state or not. The file is created before creating the State files and deleted when they are both complete. When recovering, if the monitor finds one of those files, it will read the .OLD state monitor and ignore the current one (which may not exist, be corrupted or simply not synchronized with the thread state).

Rundown - Porter

```
method() {
    Set and Save State;
    Change Variables;
    Invoke Monitor A Method;
    Change Variables;
    Invoke Monitor B Method;
    Set and Save State;
    Invoke Monitor A Method;
    Change Variables;
}
```

- No problem
- Repeat changes
- Monitor realizes
- Repeat changes and request
- Read .OLD state
- Start from the new state
- Repeat changes and request of the new state

Rundown – TD Zone

Regarding his state, it will be loaded when the monitor starts. If the TEMP file exists, it's loaded from the OLD MonitorState. If not, then it loads the regular MonitorState. If none exist, then the monitor is not recovering from any fault, it is simply being started.

While executing method calls, the monitor will save the thread states and the monitor state. After leaving the critical region, it will answer the client and delete the MonitorThreadState. If it crashes before it replies, then the client will timeout and repeat the request. If it crashes after it replies, it's no problem. The client won't invoke the method again and there are no inconsistency problems, since the monitor state has been saved.