CS7050 Individual Programming Project: Final Report

Due on Wednesday, 4 December 2013

Balazs Pete

Contents

Introduction

This report presents a prototype of a multi-hop train journey booking system, implemented in the Java programming language. The system uses stream socket communication between the different nodes in order to achieve its overall goal. The report will detail the architecture of the system, and the different approaches used to solve the presented challenges such as data handling, node concurrency, and data replication.

The system is required to allow travel agents to perform on-line booking and cancellation of multi-hop train journeys. The design of the system may ignore security and availability concerns, and may assume that nodes and disks cannot fail, however the possibility of communication failure has to be accounted for. The prototype only has to provide a rudimentary user interface for clients.

In the following sections, the architecture and the workings of the implemented system will be introduced, in addition some of the weaknesses will be highlighted. Finally, a conclusion will review the presented system and mention some steps which could be applied to alter and improve the current system for the better.

The system

The Architecture

The distributed nature of the system implies that functionality should be partitioned into separate components. The scalability property may not only refer to the number of users the system has to support, but also the amount of processing it may need to do on the data. The implementation allows for multiple train companies and for easy extension of the network, either by the addition of an another company or more train routes.

The implementation considers that a train network may be managed by several companies; with this structure, it is helping to create a better breakdown of functionality based on data behaviour. The approach taken partitions the data based on companies and allocated each partition to a separate node cluster (a set of nodes corresponding to one main functionality). This structure reduces the number of possible bottlenecks within the system. Clients are aware of the location of each cluster, the exact routing technique is described in a follow-up section.

The data may be further partitioned based on functionality and frequency of updates. Train networks (stations) and routes (set of links between stations) do not regularly change, hence they could be stored and served by a separate cluster, while other cluster would be responsible for serving the up-to-date information that is constantly changing and be responsible for the management of the booking operations.

As the project is implemented in Java, one can assume that the terminals the travel agents use to run the client application have the computational power to run some of the key processing locally, mainly the ones specific to a clients needs. An architecture overview diagram may be found in Figure 1.

The Static Data Cluster

Static data within the train network refers to the information regarding *Stations*, *Routes* and *Sections*, and a description of the system state. This information does not need to be real-time and hence it is updated in longer term intervals. This information is not stored in a database, instead is loaded from a JSON file and kept in memory then replicated between the nodes of the cluster.

The system has one Static Data Cluster, which is responsible for collating the static data from all sources and to serve it to the clients. This cluster contains several nodes, partitioning the overall functionality. The cluster has three input sources: the **network data** (stations and sections), the **mapping of a route** to a dynamic data cluster, and the **routing information** for each dynamic data cluster.

The master node is responsible for collating information, and process them in order to put them into a more optimal format. It creates four main data sources: the dynamic data cluster **routing information** for each cluster, the **mapping of route** to dynamic data cluster, the set of **stations**, and the set of **sections**.

After preprocessing the data, the master node makes it available to the slave nodes by accepting communications. Clients ask the master node for data by the means of a *DataRequest*, to which the master replies with a *DataTransfer* which contains the requested data. Once s slave has a local version of all static data, it sends a *HELLO* request to the master to register itself on the cluster. The master acknowledges the registration by replying to the message. Once registered, the slave node becomes visible to the clients. To refresh their data, slaves disconnect from the cluster, copy the data, and then reconnect to make themselves available again.

When clients require static data, they send a *Cluster Hello* request to the master node of the static data cluster. The master replies with the set of addresses for the registered slave nodes. The client chooses a slave randomly and sends data requests to the chosen slave in order to retrieve its required information. Data requests are a special messaging protocol; messaging is described in a followup section.

The Dynamic Data Cluster

Company specific data is of two different types: train route data and ticket booking data (both dynamic and changing real-time). Each company within the system can have its own separate data cluster, which implies that there may be several company clusters in the system.

The static data cluster processes a manually curated list of addresses which correspond to the location of the master node of each company's master node, but since clients are only allowed to connect to slave nodes, the static data master needs to retrieve the list of slaves for each dynamic data cluster. It achieves this by sending a *Cluster Hello* message to each master node, and just with the case of the client application, a list of addresses for the slaves is returned.

The dynamic data cluster possesses two key nodes, the master and the data store. The data store is responsible to save the data received from the master node to a file, and to provide means for nodes to be able to restore the data by serving the contents of the file. The data transfer and retrieval processes are identical to the static data scenario.

The master node is responsible to periodically transfer the contents of its memory to the data store. In addition, it is responsible to curate the list of currently active slave nodes and to allow their registration on the cluster. In addition, the master allows the receipt of Cluster Hello messages from the static data master.

Slave nodes are responsible for the communication between the cluster and the Clients. When a slave is initiated, it first contacts the data store and retrieves the most up to date save in order to ”restore” itself. In addition to the memory data, it will also request the list of currently registered nodes on the cluster. The slave uses the node list to send a Hello request to each one of them in order to let the others know it is ready and alive.

The slave allows different type of requests from the Clients, such as pre-booking, prebook delete, booking, cancelling, and status check. Any modifications on the data is applied in parallel on all nodes within the cluster (except the data store) with the use of distributed transactions.

the nodes of the cluster deal with one data collection, that is the set of *BookableSections*. These are entries identical to the sections, however contain additional fields to support the booking of seats. Each entry in the set is wrapped by a data vault, which ensures that any modification to the data are buffered and need to be confirmed prior to finalising them. In addition, the main data collection is in itself locked to ensure concurrency within the alteration of its structure.

The Client

When initiated, the client will require the static data in order to be able to do searches and reservations, however to be able to contact slave nodes within the static data cluster, it needs to retrieve the location of the slaves from the master node[1]. Once the data is retrieved, the client creates a graph from the set of stations and sections in order to be able to apply Dijkstra’s shortest path algorithm to find journeys within the train network. The user is able to select an origin and destination station, and a desired start time. From this information the client can determine the cheapest or fastest journey from origin to target and presents the first journey that is available (in time). In addition this search is able to find a path with the least number of hops within the network, if the user decides to prefer this option.

When the user has selected a journey, they are presented with the option to book it. If the user proceeds, the client will divide up the data into partitions corresponding to each dynamic data cluster. For each partition, the client contacts the corresponding cluster and requests a pre-book on each section. If the pre-book was successful, it proceeds on to the next partition. If all sections have been pre-booked successfully the client proceeds to book the sections in a similar way, however if some of the sections failed the client cancels the booking and requests a pre-book delete from the clusters for each successful section. Once a booking has been established, the user is presented with the booking information.

The involved two step booking mechanism, is in its way a transaction, as seats have to be reserved prior to finalising a booking. The booking process is cancelled in case of a failure and changes will be reverted. In addition to booking, the client is able to cancel a set of booked seats as well as get status information in sections.

The Components

Data Objects

There are two main types of data structures within the application, these objects may describe entities that belong either to the *train network* or the *system*.

[1]The process of how the Client retrieves the data is described in the Static Data Cluster Section

The *train network* describes the semantical data of the system, which is the data that users care about and what the system is here to manage. The *Network* is a graph based representation of *Station*s and *Section*s, which are objects corresponding to vertices and edges in a traditional graph data-structure. Each section may be represented as a *BookableSection*, this object contains *Seat*s that may be pre-reserved and then booked; clients do not have access to the detailed information of sections, and are only presented with the more simplistic representation, they are unable to access seating information directly. A *Route* corresponds to the traditional sense of a train route, a list of sections from station A to station B; it possesses a unique identifier.

The *system* package describes objects that are required in the handling of the operations for the system, and are used by other components such as transactions. The *RouteToCompany* object is simply an implementation of the *Map.Entry* object which maps a route id to a dynamic cluster name, this information is used to determine which cluster to connect to to book a seat for a section. A *NodeInfo* object contains a description of a system node, which contains the name of the node and its address. The *ClusterInfo* object builds on top of the node information and may contain more than one address; when an address is requested from a *ClusterInfo* object, it will randomly select and return one it contains. The *Ticket* object simply describes a booked journey information, it contains the start and end vertices, the start time, and most importantly, the set of booked seats.

Protocol Based Messaging

Node intercommunication is achieved by the use of stream sockets. Sockets are not exposed to the rest of the code, instead they are wrapped with a custom communication library which relies on messages to communicate between nodes. The library ensures a reliable message delivery, hence the the other part of the system do not have to worry about handling of message errors (this excludes a communication exception when the connection is dead). Messages are objects which may contain any Java serialisable object. These messages are used to express requests or transfer data, in addition the same messaging system is used for managing distributed transactions.

The communication system has two end points, a client and a server. It is assumed that clients will always initiate the connection, therefore establishing and severing of the connection is controlled on the client side. When a connection is open, clients can send and receive messages. In order to be able to create a connection, the node which initiates the connection will need to acquire a lock on the communications, prior to proceeding with the communication.

Servers use an approach similar to asynchronous event based techniques to handle incoming messages. Each server has a set of protocols which are instructions on how to handle a specific message type. When receiving a message, the server determines if it is capable to handle the type of the message by searching for the corresponding protocol in its list. If found, the server passes the message to the protocol and instructs the protocol to process the message. The protocol may generate a reply message, in which case the server sends this reply to the client. In case an exception occurred during the processing of the incoming message, the communication server will reply to the client with an error message. If no associated protocol is found for the incoming message, the client is let known by the means of an error message.

Data Management and Synchronisation

Described in a previous section, data within the Client node is organised according to its semantic meaning. However within the data clusters, data is stored in collections, where we have one collection for each data type. Within the static data cluster, no restrictions on the data have been put in place as all modifications are executed by local threads, however to ensure concurrency all accessors are synchronised on the data collection, eliminating any parallel access and modification on the data.

Data within the dynamic data clusters are constantly updated and replicated between nodes, hence one must ensure that there is correct data synchronisation and concurrency between the involved nodes. Any requests that are made to a slave node which modifies the data have to be immediately synchronised on the other nodes. This is achieved by the use of distributed transactions. These transactions are implemented with an asynchronous approach, where the transaction coordinator is an object instead of a thread. Whenever an event occurs, the object is woken up to handle the input event.

When a client submits a data modification request, such as a booking event, the message handling protocol creates a transaction coordinator which is then initiated in a separate thread. This coordinator will first retrieve the data vaults that contain the involved *BookableSection*s then request a write lock on the elements. Data vaults are locks which allow modifications on the contained data that are not immediately reflected. To apply any modification to the wrapped data, the changes within the vault have to be committed.

Once all data have been reserved, the transaction retrieves the transaction content appropriate for the request. A *transaction content* is an object which contains the request logic. The coordinator executes the the transaction locally. If successful it will enter the ready to commit stage, and will then contact all other registered nodes within the cluster and forward a copy of the transaction content to them. While waiting for replies, the coordinator is put to sleep.

Whenever a transaction execution reply message is received, the coordinator is waked up to process the message. If it determines that all contacted nodes have replied it will make a decision whether to commit or reply based on the success rate of the transaction on the other nodes. If any one transaction has failed, it will decide to abort the transaction, and the coordinator sends an error message to the client. However in case of success on all nodes, the coordinator will decide to commit the transaction and require all other nodes to do so. In the mean time, the coordinator replies to the client with the result of the transaction.

The transactions follow a two step commit protocol. Whenever a transaction is successful, they automatically enter the *ready to commit* stage and let the coordinator know. Until a *commit* or an *abort* message is received form the coordinator, the transaction keeps the lock in the data it is holding to ensure that other processes do not modify it in the mean time. Upon receiving a commit message, the transaction commits all the locks it has used during the period of the transaction in order to save any modifications to the data, after committing all vaults, it released them by unlocking. In case of an abort message, the node aborts the transaction by reverting the locks to their original state and releasing them. Once the commit or abort has been processed, the node lets the coordinator know by a *committed* or *aborted* message.

Locks and Vaults

Java's built in locking mechanisms are binding to threads, however due to the asynchronous nature of transactions, thread based locking is not suitable. Hence token locks had to be implemented. The main locking logic is found in the *Lock* object, all other variations are an extension of this one.

The lock uses a *Token* to identify lock requesters, tokens are differentiated based on their unique identifier. Each token can be of two type: read or write. A lock may have several active read tokens, however it may have only one active write token, in addition no active read tokens are allowed while a write token is active.

Each lock has a collection, which contains the active tokens. When a locking request is initiated, the lock creates a token for the request (setting the token type to the request type) and adds the token to the token queue, the accessor method blocks the requesting thread until the token has been handled. In order to ensure concurrency, the locking and unlocking methods are synchronised on the lock itself therefore ensuring that no parallel access is done. After putting the token onto the queue, the lock initiates the queue handling process.

the queue handling process is the following:

- If the lock has no active tokens, then the lock adds the first element within the token queue to the active token set. If the type of the token is write, it sets the lock to write mode. The process will check if there are more tokens in the queue, if there are, start from the beginning.
- If the lock has only read tokens
  - If the first token in the queue is a read token, add the token to the active set. If there are more tokens in the queue, start again.
  - If the first token is a write token, then don't do anything. Start again.
- If the lock has an active write token then so nothing. Start again.

When unlocking, the lock will remove the token from the active set. If the token being removed is a write token, the lock mode is set back to read.

Failure Scenarios

Deadlock within Locking Resources

If the implementation were to follow the standard token upgrade and keep until the end scenario, then deadlock scenarios would occur which would need to be handled by deadlock resolution algorithms. To avoid deadlocks of this type, we do not allow lock upgrade and require processes to determine what type of locks they will require prior to execution.

A possible deadlock scenario with the current implementation is in thread deadlock. This would occur when a thread requests a read lock on an object, then requests a write lock (or vice versa) without releasing the first lock prior to the second request. This is due to the way locks are implemented, as a lock request will block until the suitable condition will arise, however it never will as the thread cannot unlock itself due to being blocked.

Communication Failure

A possible scenario is the failure of communication between two nodes. The communication of the system has been presented in the previous section, and has been mentioned that the library handles communication failure. Possible scenarios are the break of the TCP connection between two nodes, and the inability of establishing or severing a connection. We do not have to worry about integrity check or reliable messages, because given we have a connection between two nodes the TCP protocol takes care of all these aspects.

When initiating a connection, the messaging client will try to establish a TCP connection to the messaging server, if it is unable to make a connection it will retry N amount of times. If no connection has been made within the given window, a communication error is raised, which is handled by a higher level code. In most cases, the higher level code would consider the node we wish to connect to unreachable and forgets it. If an error occurs during the sending of a message, the library will retry several times prior to raising an exception.

When the client is excepting a message from the server and an error occurs, the client will resent its message to the server before waiting for a reply. The server on the other side will handle this failure. It will determine that the incoming message is a duplicate, and it will server the response of the previous message.

If an error occurs when the client wishes to severe connection from the server, it will try again several times. However if at the end it still fails, it will ignore the error and continue with the execution.

Possible Functionality Failure while Scaling

Within clusters, if we have a high number of slaves, the master node might become a bottleneck causing the entire cluster to slow down due to wait times. The volume of slaves that might cause this issue is if unknown. Clusters were tested with up to four slaves without any loss of performance.

Delayed Responses from the Dynamic Data Cluster while Scaling

The response time for booking requests might increate significantly as the number of nodes within a dynamic data cluster is increased. The cluster has been tested on a configuration of four slaves without loss of performance. The possible failure scenario would occur in a cluster of a significant size (hundreds of nodes). Additional delays would be introduced by the increased number of concurrent client requests. A possible approach to reducing the load and circumvent the issue is to divide the data the cluster is responsible for between the old cluster and multiple new clusters.

However it is important to note that we may have transactions that are executed concurrently within the cluster, given that they do not use the same resources. In addition, each node is able to coordinate multiple transactions at the same time.

Failure of Key Nodes

A possible failure scenario would be if a master node within a cluster would to fail. If this would happen both the static and dynamic data clusters would cease updating, main functionality would work however they would not be able to scale or refresh with new data. However the implementation assumes that nodes do not fail therefore no focus have been given on resolving such issue.

Identical Nodes on the Same Hardware

Since nodes use stream sockets for communication, whenever a socket is created, the application binds to the port on an OS level, reserving it and making it unavailable for other processes to use. If we have two nodes on the same machine which communicate on the same port, we will experience communication exceptions as one node will be unable to reserve the required resources in order to work. To circumvent this issue, we assume that two nodes of identical functionality will not reside on the same hardware.

No stable connections

Currently, for every request a TCP connection is opened then closed. This can be very expensive is we have a lot of concurrent requests to the same node. An approach would be to use the same connection to send multiple messages concurrently to reduce the latency. Unfortunately, the current communication client implementation does not allow for such use case and would need some redesign.

Security

Given that security implications were not in focus when implementing the system, the current implementation has some key security holes that would not be acceptable in a ready to deploy system.

- Communication is unencrypted and hence anyone can see the transferred data
- Node registration is unrestricted and anyone knowing the registration protocol may join
- Transactions within the dynamic clusters nodes make use of remote code runs, given that communication is un-secure, anyone mocking a transaction may alter the data of a node without any restriction

Conclusion

In this report, a prototype of a multi-hop train journey booking system was presented. The main functionalities and components of the system were highlighted, and in addition the general process of the system was described with a focus on data concurrency.

The architecture for the three main components of the system was discussed detailing the management and flow of the data within the node, and how it is transferred between nodes. distributed data management techniques using transactions were presented. Finally, low level components were discussed in terms of how they relate in the overall system.

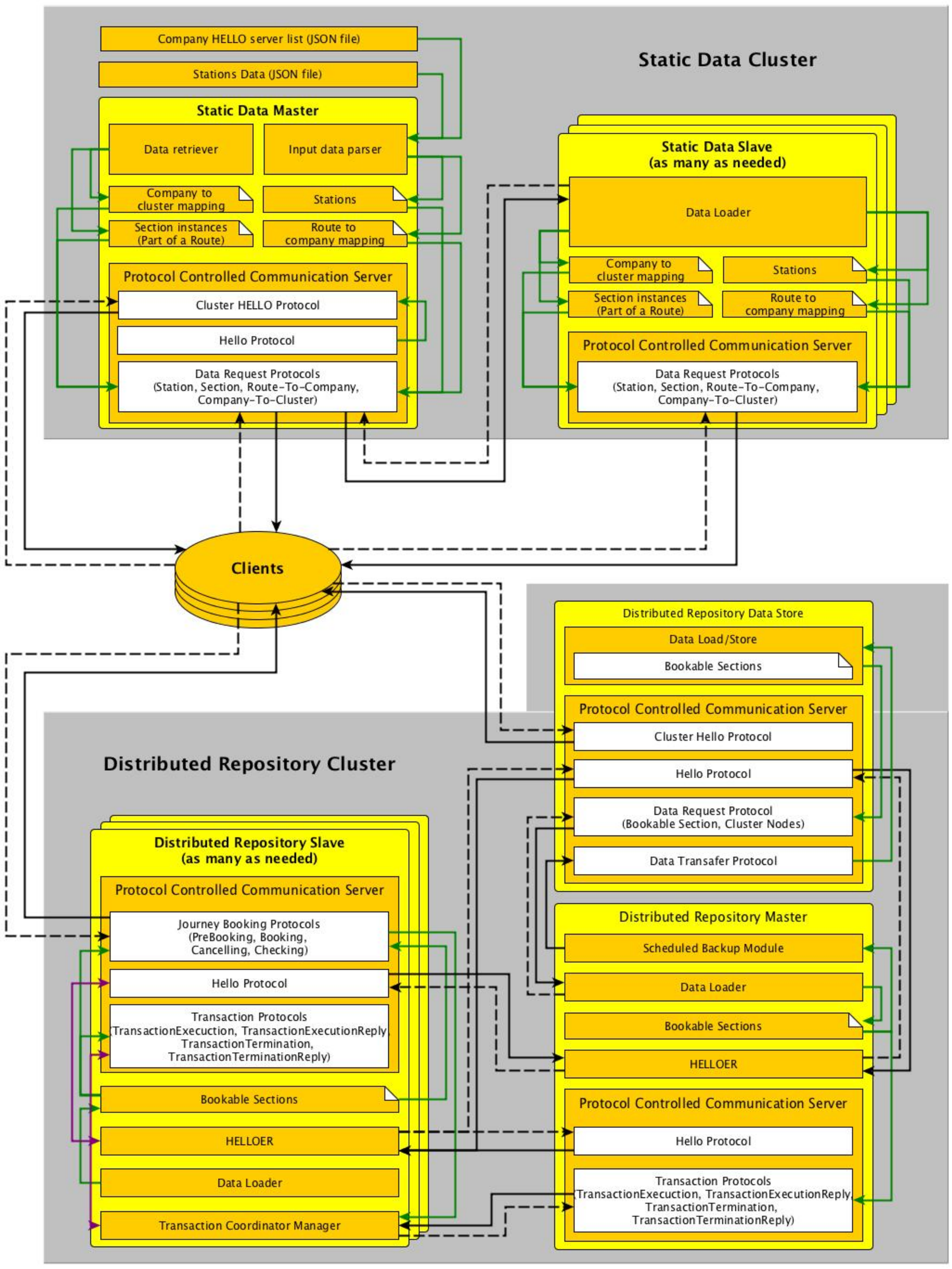
Static Data Cluster
Company HELLO server list (JSON file)
Stations Data (JSON file)
Static Data Master
Data retriever
Input data parser
Company to cluster mapping
Stations
Section instances (Part of a Route)
Route to company mapping
Protocol Controlled Communication Server
Cluster HELLO Protocol
Hello Protocol
Data Request Protocols (Station, Section, Route-To-Company, Company-To-Cluster)
Static Data Slave (as many as needed)
Data Loader
Clients
Distributed Repository Data Store
Data Load/Store
Bookable Sections
Cluster Hello Protocol
Data Request Protocol (Bookable Section, Cluster Nodes)
Data Transafer Protocol
Distributed Repository Cluster
Distributed Repository Slave (as many as needed)
Journey Booking Protocols (PreBooking, Booking, Cancelling, Checking)
Transaction Protocols (TransactionExecuction, TransactionExecutionReply, TransactionTermination, TransactionTerminationReply)
HELLOER
Transaction Coordinator Manager
Distributed Repository Master
Scheduled Backup Module

Figure 1: The Architecture Overview - black: inter-node communication, green: intra-node data flow, purple: inter-node communication between nodes of the same class