FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Distributed Systems

3rd year of the Integrated Masters in Informatics and Computing Engineering (MIEIC)

Adversarial Search Task Distribution via Chord Protocol

T5G07
André Cruz - up201503776
António Almeida - up201505836
Diogo Torres - up201506428
João Damas - up201504088

28 de maio de 2018

Table of Contents

Introduction

The developed project consists of an application capable of distributing adversarial search computations. Adversarial search algorithms become computationally heavy fast and, when executed on live applications, which is one of their main use cases, can become very impractical: we intend to obtain fast results, but the calculation times grows exponentially. Take Minimax for an example: if we consider a game state tree with a branching factor of *b* and an average game length of *m* moves (or a lookahead factor of *m* moves), its complexity is estimated to be $O(b^m)$. In games like chess, where b ~ 35 (per move) and engines often need to look at least 30-40 moves ahead, this becomes extremely computationally expensive.

With this in mind, the application implements a peer-to-peer system through the Chord protocol. When asked to compute such a task, the application partitions it in sub-tasks (e.g. in the case of Minimax, sub-trees) that later distributes to other peers (which can recursively repeat the process given a still complex enough task).
In the next sections, the application will be described in finer detail, namely its architecture, implementation details and relevant issues. This report concludes with a small reflection about the work done and possible future enhancements.

Architecture

The application consists of a peer-to-peer network, implementing a distributed hash table through the Chord protocol. The peers communicate between themselves through secure TCP sockets, and the user can interact with peers via an RMI interface.

The remote interface enables the user to get the status of the peer it is connected to, to lookup data on the distributed hash table, to store data on the distributed hash table, and to request the execution of adversarial search tasks. All these methods require authentication, with the users' usernames and passwords stored on the chord network itself. Note that any adversarial search task can be submitted for execution, as long as properly defined using the given *AdversarialSearchProblem* interface. As expected from an adversarial search problem, this interface expects the implementation of three methods: a *successors* function, to retrieve a state's successor states; a *utility* function, to retrieve a state's perceived value; and a *terminal* function, to check whether the given state is a terminal state.

The peer network revolves around the Chord protocol. This protocol uses a distributed hash table (DHT) to help in the resolution of unstructured names in a network. A DHT essentially stores pairs (key, value). A set of keys is associated with a given peer/computer, known as a node. This node stores the values of all the keys for which it is responsible. Both nodes and keys are assigned an identifier with a predefined value of m bits using consistent hashing, reducing the average amount of keys that need remapping after the table is resized. The nodes (and keys) are then arranged in a circle with, at most, 2^m elements. Each node has a predecessor and a successor, which are the first non-empty elements in the circle counter-clock and clockwise, respectively. No node is more important: all are equiparate and equivalent, providing a decentralized service.

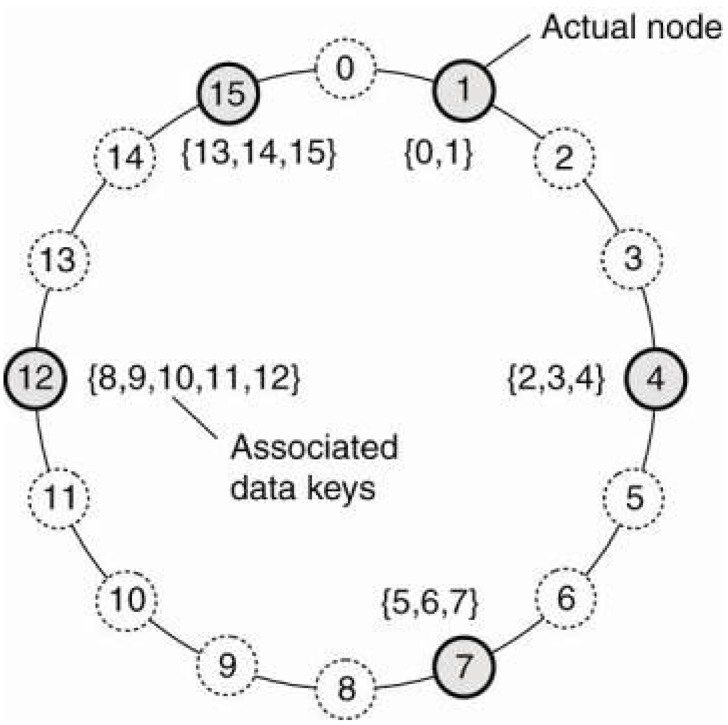

Figure 1 - An example of a chord ring

The chord protocol specifies how keys are assigned to nodes and how a node can lookup the node responsible for a given key. In fact, the lookup operation is the main goal of the protocol. With this in mind, the protocol also specifies that each node must contain a finger table with, at most, m entries. The n-th will represent a node that dists 2^n (mod m) from the node, so the first entry is actually the immediate successor. Every time a node wants to look up a key, it will pass the query to the closest successor or predecessor of k in its finger table. This process is then recursively repeated until a node finds out the key in an immediate successor. We can then conclude that the presence of a finger table will reduce, in each iteration, the search space in half. Therefore, the lookup operation will have complexity O(log N), if N represents the number of nodes in the circle.

In order to guarantee consistency in the pointers and, therefore, lookup operations, a stabilization protocol runs periodically, updating finger tables and successor pointers. This way, node entry and exit in the circle can be done mostly with little to no problem, by appropriately notifying nodes. Additionally, a handoff task is performed periodically, which checks whether the current node holds any data which he is no longer responsible for, handing off that data to its new owner node.

The communication between nodes is done via a serializable message protocol defined by the group. Each message contains the following information: id (in order to match request and response messages), type (e.g., predecessor/successor request, task fulfillment request, lookup/store request), possible argument (e.g. in case of task fulfilment, the task’s essential information), serializable data (in case of a store request), a boolean representing its state (whether it is a request or a response message), and the sender’s address (to facilitate sending responses). Two possible request types are possible: for synchronous requests, two sockets are opened (one for input, another for output), with the messages being exchanged in them. For asynchronous requests, the request message is sent and a callback function is registered with the request’s id. The callback function, which receives a response Message as argument, is triggered when the corresponding response is received.

Implementation

The whole application was implemented using the Java programming language and no external libraries/frameworks.

The communications between the networks' peers is assured by the use of secure TCP sockets, using the *SSLServerSocket* and *SSLSocket* interfaces. Regarding the communication protocol itself, it is implemented by the use and serialization of *Message* instances, containing the type of the message, whether it is a request or a response, and the message's arguments (*e.g.* the task's definition). The handling of message requests is processed in the *MessageDispatcher* class, providing several helper functions for common requests, as well as handlers for all types of messages and corresponding responses.

Thread safety is ensured by the use of Java's *Concurrent Collections,* from java.util.concurrent, namely *AtomicReferenceArray* for the peer's finger table and ConcurrentHashMap for the peer's in-memory Serializable data.

Concurrency and parallelization is achieved by the use of several thread pools, one for processing adversarial search tasks, one for listening for incoming messages, and another one for dispatching requests. Additionally, the peers' task handling functions return a *Future<Integer>* object, corresponding to the task's value, allowing for complete concurrency of operations and later retrieval of all results.

On the client's side, concurrency is ensured by the use of asynchronous requests, registering a callback function alongside the request (*sendAsyncRequest* in lines 93-97 of the *MessageDispatcher* class). This callback function is later triggered when the request's response arrives (*handleResponse* method in lines 148-158 of the *MessageDispatcher* class).

Additionally, all helper threads are ran in parallel on separate threads. Those which require continuous recurring tasks extend the *RecurrentTask* interface, which receives an interval in milliseconds and runs the child's *run* method periodically, facilitating boilerplate for repeating tasks.

Finally, each peer exposes a remote interface for use of its services, including the hash table's customary *put*/*lookup* methods, as well as the *initiateTask* for our particular application. This remote interface is defined in the *RemotePeer* interface, and the *RemotePeerImpl* class provides a wrapper around the Peer interface which exposes the appropriate methods to the client.

Relevant Issues

Scalability

The chord protocol distinguishes itself from its predecessors for precisely ensuring scalability. By keeping information about only a few amount of nodes (around log N, if N represents the amount of nodes in the network) and with automatic finger table adjustment on node entry or exit (through notification messages), the protocol ensures the number of nodes can scale up quite well.

Security

Security in the application is ensured through the usage of Secure Sockets provided by the classes SSLServerSocket, SSLServerSocketFactory, SSLSocket and SSLSocketFactory, all from the javax.net.ssl package. Usages can be seen in the Listener and MessageDispatcher classes (both in the network.threads package), e.g. the server socket is initialized at Listener.java, l.38 (beginning of respective function). The client socket is initialized at MessageDispatcher.java, l. 124 (beginning of respective function). The keys used were the ones provided in Lab 5.

Alongside them, a login mechanism is used to ensure no unwanted accesses occur. When initiating the client, the user must authenticate before he can progress any further (service/InitClient.java, l.62). This login process consists of retrieving the credentials from user input and testing them against credentials stored in the peer (service/Authentication.java, ll.11-18 & 35-50).

Fault tolerance

Fault tolerance is another property ensured by the chord protocol. Through the usage of stabilizer threads, which periodically pings the node's successors so as to assess if they're still operating, the node can adjust his finger table, replacing eventual entries to nodes that failed to report as (still) alive with its nearest successor that is. Additionally, the node's stabilization protocol also ensures the network's structure is kept consistent, as it continuously checks whether the node's position in the network is valid.

Conclusion

With this work, the group realized the importance of a well structured architecture in the development process in order to achieve a good final result. The final application provides an easy and efficient way of distributing traditionally heavy computational work, exploring different topics in the Distributed Systems area. For this reason, the opinion is that a good project was developed.

Despite this, as always, there can be enhancements done so as to further elevate its quality and performance. In this case, the extensive use of asynchronous requests even for fast-fulfilling requests (*e.g.* pings and successor/predecessor queries) could provide an extra step in enhancing the parallelism of the application.