author: 170025929

CS4103 Practical 2 - Distributed Entry Nodes

Implemented Functionality and Extensions

- Random network generation using provided hosts
    - Network Visualisation
- Automated network parsing and self-identification for each node
- Automatic start of election process if no leader is present
- Automatic election of a new leader if the leader node dies
- If a leader cannot reach more than half of its followers, it shuts down to prevent two leaders in separated network segments if the network was split
- Leader election using the wireless election algorithm presented in lecture 11, week6
- Logging
- Logging to a central server using the syslog protocol
- Regular leader broadcasts to prevent new/restarted nodes from starting a new election
- Heartbeat messages for crash detection
- Resource access control by the coordinator using a token valid for max 2s
- Token times out if a client crashes
- If multiple resource access requests are queued, these are combined into a single request.
- Resource release once a client finished using it
- Maximum visitor entries, all ticket numbers of visitors currently in the park are stored in a shared sqlite database (performance over index free database)
- Visitors can enter and leave the park at any node
- Client: simple UDP interface

Design

Network Parsing/Node Setup

Each node is given a network graph (format: graphviz dot) that specifies the possible connections between nodes and the machine and port a node runs on. During

initialisation, each node parses the network graph, uses the hostname of the machine its running on to identify which graph vertex it is represented by, and sets up connections to all reachable nodes in the network. This setup method has been used to be able to start each node with the same command using automation tools like pssh. However, this also means that only one node can be run per machine, which I considered not to be a problem as not forbidden in the practical specification.

Implementation: `networkParser.h` `networkParser.cpp`

Inter-Node Communication

To reduce overhead, the nodes communicate using UDP connections. The communication subsystem uses fixed-sized messages of different types that can have a payload. The currently used and implemented message types are : `HEARTBEAT, TEXT, ELECTION, ACK, COORDINATOR, LOCK, UNLOCK, GRANT` . For each message type and connection a callback can be registered which will be executed when a message of the specified type arrives from the specified remote. Message handling is performed on one asynchronous background thread using udp socket operations and boost::asio for asynchronous operations support.

Implementation: `connectionManager.h` `connectionManager.cpp` `networkMessage.h` `networkMessage.cpp`

Leader Election

Leader election is performed according to the wireless election algorithm as presented in slide set 11, week 6. Once a node is elected as leader, it establishes ephemeral connections to all other nodes (as required by the specifications of the assignment). Those connections will automatically be removed if they time out or if a new leader is elected. A unlimited number of sequential elections is supported. While the system has not been designed to support concurrent elections, they seemed to work fine in my tests (when several nodes are started at once, e.g. by my runPSSH script, they sometimes start several concurrent elections).

Implementation: `election.h` `election.cpp`

Access Control

Node Resource Access Control

If a node need to access the visitor database, it needs to request a resource lock from the current coordinator/leader by sending a `LOCK` message. The leader will reply with a `GRANT` message, either immediately or once the current resource lock expires. The lock is

valid only for a defined time period (current implementation: 2 seconds) or until it is released by the node that acquired it (`UNLOCK` message sent to leader from current lock holder). Implementation-wise all resource requests provide a callback that gets executed once a `GRANT` message is received. If several requests are outstanding, the provided callbacks are combined to be executed in a single request. If an election is in progress, no resource requests are possible and will be rejected. If the leader changes while a resource request is outstanding, the outstanding requests are cancelled. This is possible in the designated use case as the customer can simply rescan their ticket a second later.

Client Access Control

A sqlite3 database is used to store all current visitors of the park and the maximum permitted number of concurrent visitors. When a client presents a ticket number, database access is requested by the corresponding node and, if the ticket is not currently in the park and the number of current visitors is smaller than the permitted number of visitors, the ticket id is added to the database and access is granted to the visitor. If the park is full or an election is in progress, the visitor is asked to retry later. When leaving, the node requests a resource lock, checks if the ticket exists in the database and removes it if it exists. A sqlite database is used instead of a simple file for performance reasons. The index structure of a database allows for significantly more efficient access once a few dozen visitors are in the park. The visitor limit is stored in the cfg table, it can be changed there or in the visitorAccessMgr.cpp file. *Note:* The database is persistent, i.e. the visitors in the park will be remembered after a system restart.

Implementation: `lockManager.h` `lockManager.cpp` `visitorAccessMgr.h` `visitorAccessMgr.cpp`

Fault Tolerance

All connections are monitored using a regular heart beat message with interval and timeout specified in `connectionManager.h` . To reduce network traffic, heartbeat messages are only sent if the connection has been idle for >= HEARTBEAT_INTERVAL. On the receiving side, all messages received by a remote host fullfill the heartbeat functionality. The current heartbeat interval is 2 seconds, the heartbeat timeout is 2.5*2 seconds to account for possible connection and processing delays.

Implementation: `connectionManager.h` `connectionManager.cpp` `election.h` `election.cpp`

Logging

Logging is implemented using boost::log with different sinks for console output and syslog-compatible (RFC5424) network logging. The severity levels `trace, debug, info,`

`warning, error` are supported. On stdout all messages of severity >= debug will be shown. The syslog backend will send out all messages as filtering should be done server-side when using a network logging system. On stdout the node id is omitted as all output clearly comes from the single node instance running on the machine. The severity level is also omitted to provide log output similar to the example given in the practical specifications: `[2018-04-23 15:13:11] - Election: received ack message. from pc5-003-l:12345` .

The network log sink include host, timestamp, component and message: `(shortened) Rfc5424SyslogEvent [level=7, timestamp=15:22:34, appName=pc5-021-l.cs.st-andrews.ac.uk, structuredData=Connections:, message=Received heartbeat from pc5-021-l` . *Note:* level 7 equals log level trace.

Implementation: `logging.h` `logging.cpp`

Client Interface

Each node provides a simple, text-based interface to one or multiple clients (i.e. ticket/entry gates) via a UDP connection to port `1234` to the machine the node runs on. The interface allows to request access for a ticket number (`e ticketID` , e.g. `e 1234`), request exit permission (`l ticketID`) which will only be granted if the ticket was previously used to enter the park through any entry gate, list all current visitors (debug functionality, not protected against race conditions, won't lock database; `v`), terminate the node (`q` or `c`) and provides functionality to start an election (debug, not needed as the system will automatically elect a new leader if required; `E`). A ticket ID may only be used to enter the park once until it is used to leave the park. Before each command a newline must be entered.

CLI

The syslog server, network graph file and visitor database location must be specified as command line arguments on program execution.

Examples and Testing

The nodes were continuously tested during development on randomly generated network graphs of 3 to 25 machines. As the network layout changes every time a network is generated, the large number of test runs performed during development should suffice to cover all realistically occurring edge cases. As the nodes need to be run on different machines by design, unit testing becomes very complicated and thus is not done on this project.

Example 1: Node receives election messages from several connected nodes

```
[2018-04-23 15:13:11] - Election: received election message. Setting parent t
[2018-04-23 15:13:11] - Election: received election message. Already has pare
[2018-04-23 15:13:11] - Election: received ack message. from pc5-003-l:12345
[2018-04-23 15:13:11] - Election: received ack message. from pc5-002-l:12345
[2018-04-23 15:13:11] - Election: received acks from all adjacent nodes. Senc
[2018-04-23 15:13:11] - Election: highest received metric: pc5-001-l:12345:61
[2018-04-23 15:13:11] - Election: received new leader: pc5-001-l:12345
[2018-04-23 15:13:11] - Election: I am the new leader
[2018-04-23 15:13:11] - Post Election: Starting ephemeral client connections
[2018-04-23 15:13:11] - Main: leader changed
[2018-04-23 15:13:11] - Checking and preparing database after leader change:
[2018-04-23 15:13:11] - Election: broadcasting new leaderpc5-001-l:12345
```

Example 2: Node starts election and broadcasts new leader

```
[2018-04-23 15:13:11] - Election: starting election
[2018-04-23 15:13:11] - Election: received ack message. from pc5-003-l:12345
[2018-04-23 15:13:11] - Election: received ack message. from pc5-001-l:12345
[2018-04-23 15:13:11] - Election: received acks from all adjacent nodes. Elec
[2018-04-23 15:13:11] - Election: highest received metric: pc5-001-l:12345:61
[2018-04-23 15:13:11] - Main: leader changed
[2018-04-23 15:13:11] - Election: broadcasting new leaderpc5-001-l:12345
```

Example 3: Lock and Unlock request

```
[2018-04-23 14:45:03] - LockManager: received lock req from node pc5-004-l:12
[2018-04-23 14:45:03] - LockManager: Sent grant to node pc5-004-l:12345
[2018-04-23 14:45:03] - LockManager: received unlock req from node pc5-004-l:
```

Example 4: Less than half of the network reachable by master, syslog

```
>>> Syslog message came: Rfc5424SyslogEvent [prioVersion=<10>Apr, facility=1,
appName=pc5-003-l.cs.st-andrews.ac.uk, procId=pc5-003-l, msgId=:, structuredD
message=Lost connections to majority of network. Shutting down to avoid netwc
```

Example 5: Client Interface

```
pc5-002-l:~/dev/simple-syslog-server lw96$ nc -u pc5-001-l 1234

Select action:
c: crash node
q: quit node
E: start a new election
e <ticketID>: new entry
l <ticketID>: leaving visitor
v: print current visitors
```

Evaluation

Functionality implemented as described above. Potential improvements:

- Kill concurrent elections as they can cause unnecessary traffic and potentially undesired behaviour
- More advanced access control, e.g. resource blocks
  - Not implemented as it would be complicated to decide which tickets entered the park if a client crashes while it holds a lock for a number of tickets. I implemented the combining of requests instead.
- Flexible Network
  - Would allow a more dynamic system; however, it would be hard to decide if a master is connected to a majority of the network. This is important if all central wireless nodes fail and the network splits. If both network segments elect independent leaders, this can lead to multiple clients getting a resource lock and accessing the same resource concurrently.

Used Libraries

Libraries are included as submodules from their GitHub projects and thus not additionally linked here.

- boost::asio
- boost::log
- boost::graph
- SQLiteCPP
- sqlite3
- pssh
- simple-syslog-server
- python igraph

- python pycairo

Build System

CMake

The node developed for the practical entirely relies on CMake as build system. It can be build by simply running `cmake /pat/to/project` and `make`. All required libraries will be downloaded, compiled and linked as necessary. As the specifications require the project to ship all libraries that are being used, the compilation can take several minutes as several boost components and the sqlite3 library need to be build. As several C++17 features are used in the code, an up-to-date CMake and gcc installation is required. The clang compiler should also work, although extensive testing was only performed using the gcc available on the school's Fedora machines.

Note: The code relies heavily on the boost library, the template resolution issues of several gcc and clang versions can cause it to use up to 3GB RAM during compilation.

Python

As I did not want to manually create several different, potentially large network setups for testing, I build a python script to generate randomized network layouts using the ForestFire algorithm and corresponding scripts to build and run the node on different machines. This requires the python igraph librariy to generate the network graph and the pssh module to deploy to the machines. All required modules will automatically installed into a python3 virtual environment located within the project directory.

Note: A bug in the python venv module does not allow for nested virtual environments. If the scripts are executed in an active virtual environment, the already activated environment will be used to install the required libraries.

Java

To allow for central logging of all log messages of different severities of all nodes, the syslog protocol is used. Again, as the specifications require shipping all required libraries and tools, I added a simple, standalone java-based syslog server project that is automatically build using maven. However, I recommend running a more powerful syslog system to receive and view the log messages.

Summary

Required build environment:

- JDK 8
- maven
- cmake
- gcc
- python3
- netcat

Usage

Note: The project folder must be on a drive shared by all machines.

Initial setup: `bash setup.sh`

Generate randomized network: `python3 generateNetwork.py host1 host2 host3 ... hostn`

Example: `python generateNetwork.py pc5-002-l pc5-001-l pc5-003-l pc5-004-l pc5-007-l pc5-008-l pc5-009-l pc5-011-l pc5-020-l pc5-021-l pc5-027-l pc5-029-l`

Build nodes on all machines in the network: `bash build.sh`

Start syslog server on local machine: `bash runLogserver.sh`

Run on all machines: `bash runPSSH.sh` . *Note:* requires accessible ssh key to connect to all machines and host keys of those machines in known_hosts.

Connect to the nodes to terminate them or present an enter/exit ticket: `ncat -u host 1234` and press enter

1978 words exclusive log snippets