Programação em Sistemas Distribuídos
(Distributed Systems Programming)
2016/17
MEI/MI/MSI

Project Assignment

WideBox: Design and implementation of a distributed and resilient very-large scale interactive system

1 Introduction

The purpose of this project is to design WideBox, World Wide Theatre Box-office, a logically centralized theatre box-office, with wide-scale distributed access by clients through the Internet, allowing on-line reservation and purchase of tickets for virtually all theatres available in a region, a country or even the entire world.

In order to address requirements of dependability and performance, the service implementation may itself be physically distributed, though the related details should be transparent to the users, who see a **logically centralized service**.

The **objective** of this work lies more on the distribution, concurrency, synchronisation, dependability and scalability issues of the service, rather than on the fancy aspects of the application's logic of a fully-fledged service. In consequence, we will keep the logic as simple as possible. In addition, the graphical user interfaces used in the prototype and associated tools can be simple, possibly plain text, as long as they allow to conveniently use and test the required application functionalities.

The project will be developed in three phases. They serve as milestones that help the student keep pace and fulfil the main objective. The phases are not independent, so failure of a phase will condition the following phases. On the other hand, failure of later phases still allows a partial grade to be obtained from previous phases. The objectives of each of the three phases will be completely clear after reading the whole project description, in the end of which they will be recapitulated. They are summarized below, just for planning purposes:

I. **Design and implementation** of the WideBox **full** application **logics**. **Design and implementation** of the **traffic and failure generators**.

II. **Demonstration** of the capacity of withstanding very large-scale **continued workloads**, **AND** very large-scale **peak workloads**, by **designing and implementing** the necessary mechanisms.

III. **Demonstration** of the capability of withstanding **failure of any single individual server**, by **designing and implementing** the necessary mechanisms to enhance the previous design.

2 WideBox description

WideBox is a ticket-selling service, serving as a front-end for several theatres. For instance, a client can reserve and purchase movie tickets. Several parameters of the system will be fixed in the project, for simplicity. Additionally, ticket prices and actual payment details are ignored, also for simplicity. A specification of the functional and non-functional requirements is provided in the following sections.

2.1 Functional requirements specification

The WideBox service is accessible through the web, at a well-known URL, and clients can use a common web browser to interact with WideBox. The following options, and respective service actions and replies, must be made available to a client:

- **Search** the available theatres.
  - The service replies with a ***<theatres>*** message, indicating the available theatres/movies. For simplicity, let us consider that there is only one movie per theatre, one movie session per day, and only allow reservations for the current day.
- **Query** the WideBox service about the availability of seats in a theatre.
  - If seats are available, the service will reply with an ***<available>*** message containing a map of the theatre for the (only) session, marking differently the *free* and *occupied* seats at the time of request.
  - Additionally, upon replying, the service already suggests a seat to the user, which is locked provisionally and will be seen as non-free by ulterior requests from other competing clients, but marked differently (*reserved*) from the previous two categories (*free* and *occupied*). The reserved seat is also marked in the returned message.
  - Since we expect that the client may not reply back after a seat is reserved pro-actively by the system, implementation of expiration mechanisms for unconfirmed reservations **is required**. For simplicity, in case of expiration, notification to the client, as would have to happen in fully-fledged systems, **is not required**, but the reserved seat is freed. The service must keep a separate timer for each existing reservation.
  - If the movie session is full, then the service replies with a ***<full>*** message.
- **Accept** a reserved seat.
  - If the seat is still reserved, the service commits the ticket and considers the seat occupied from now on, replies with an ***<accept_ok>*** confirmation of purchase to the client, completing the interaction.
  - If the seat is no longer reserved (namely if the reservation has expired), the service replies with an ***<accept_error>*** message.
- **Reserve** a new seat.
  - If the new seat requested is free, then it allocates (locks provisionally) the new seat, which becomes reserved, and de-allocates the old seat. In this case, the expiration time is renewed. Otherwise, nothing is done and the old seat continues to be reserved.
  - Next, the service replies with an ***<available>*** message containing a map of the theatre, marking the free and occupied seat and either the new or

the old seat as reserved, depending on the outcome of the previous step, going back to waiting a client request.

- **Cancel** the reservation.
  - If the seat is still reserved, the service will free it and will respond with a ***<cancel_ok>*** message.
  - If the seat is no longer reserved, the service will respond with a ***<cancel_error>*** message.

Each client must be identified by the WideBox service, so that it is possible to keep track of owners of existing reservations. Therefore, clients are assigned unique numeric identifiers, which are used when interacting with the service.

A typical interaction with the service is as follows. The client connects to the service by issuing a **search** operation, receiving the available theatres. Then it sends a **query** request for one theatre, receiving back the map of seats, including the free and occupied seats, along with the provisionally reserved seat. After that, the client can **accept** the reserved seat, **reserve** a different one, or **cancel** the reservation. For simplicity, it can be assumed that a client will not perform a new query while a previous query/reservation has not been completed or cancelled.

2.2 Non-functional requirements specification

The service should:

- Provide adequate consistency and performance from a QoE viewpoint (quality of experience), that is, users should witness neither inconsistency in individual transactions, nor frequent or too large delays or blackouts, in the presence of the following stress scenarios:
  A. Very large-scale **continued workloads**, even if only at specific periods of the day, due to serving clients throughout the world, for **all the theatres** included in WideBox.
  B. Potentially very large-scale **peak workloads** from clients throughout the world, for **specific theatres**, caused by very high numbers of simultaneous users competing for an event (e.g., a première).
  C. **Failure of individual servers**, both under normal and under heavy load conditions: APP or DB server (see Section 3.3). In this work, we do not address WEB server failure, there is mature technology to do load balancing of HTML requests to web server pools.
- Manage server input queue saturation of each server, as load increases and limits are achieved, in a non-disruptive way, preserving both server stability and client at-most-once semantics (that, requests cannot be lost or cancelled without the client knowing).
- Ensure **disaster prevention** in the sense of a non-catastrophic common-mode failure of many or all servers (e.g., large power outage), with regard to safeguarding the integrity of critical data: the theatres state, the memory of purchases made.

As a consequence of these non-functional requirements, the designer should ensure mechanisms that:

- Prevent concurrency, synchronization and overload issues from yielding server blocking or stalling syndromes in high-usage periods, that is,
    - when the several **server queues, for example, are about to fill up** as a consequence of too large input traffic, they should stop accepting requests and reply back with a ***<busy>*** exception;
    - likewise, in those periods, clients should always get feedback about how successful or unsuccessful their requests are, that is, **when server queues are saturated**, instead of the normal replies foreseen in the Functional Requirements Specification they should also get a ***<busy>*** reply back.
- Promote dynamic and seamless scalability to workload changes, both:
    - **long-term**, as the service becomes increasingly popular and the continued workload increases, that is, **by incremental scalability** of the infrastructure, which should thus be modular,
    - and **short-term**, as workload suddenly peaks (e.g. movie premières) from the background load, which may already be high, **by over-provisioning**, e.g., by replication of resources for performance.
- Secure consistency of individual transactions in the presence of accidental faults in servers (e.g., crashes), that is,
    - the **failure semantics seen by any client of a remote operation should at least be zero-or-once (ZoO)**, that is, "The system will confirm a purchase if and only if that purchase was completed".
- Provide for availability in the presence of accidental faults in servers (e.g., crashes), that is,
    - **server replication** for fault-tolerance should be foreseen.
- Combine both strategies, for performance and for fault-tolerance, to yield the best efficiency possible.

3 Implementation requirements

3.1 Configuration

The system database should be preformatted with **NrTh=1500 theatres**, each with **NrRw=26 rows** (A-Z) by **NrCl=40 columns** (1-40). The timeout to free reserved seats that are not accepted on time should be set to **15 seconds**. It should be possible to change these configurations easily and dynamically, e.g. during the project demonstration.

The **database should be initialised to all places "free"**.

A client is assigned a **random ID number upon start up, in the range 1-NrCl**, with **NrCl=100000**.

The replication degree (number of replicas), whenever needed, is left to the discretion of the designer, but given the limitations of the lab setting, **experiments with 3-4 replicas will be perfectly acceptable**. More can be used, though.

We consider the **single fault assumption**, e.g., only one server is crashed at any given time, any of APP or DB server (as said previously, WEB server failure is not considered, so WEB server replication is not necessary).

We consider that there is a small but non-negligible probability of **non-catastrophic common-mode crash failure of all servers**, with regard to which the **system should preserve integrity of stored data**.

3.2 Design evaluation

In order to evaluate the implementation, in particular the non-functional requirements, a simple **traffic and failure generator** tool must be designed and implemented. The tool must be capable of simulating the generation of multiple requests originating from multiple clients, for multiple theatres within WideBox.

Note that you have to design and program an algorithm that not only **generates requests** to the service, but also **reacts to its responses** in ways that do not inappropriately block the tool or the service. However, if you remember the functional requirements, this should be as easy as iterating through theatres, performing the following requests: (1) **search** the theatre; (2) if the reply message is *available* then **accept** the pre-reservation; (3) if the reply message is *full* then select another theatre. Alternatively, for query-only requests: (1) **search** the theatre; (2) cancel the reservation.

3.2.1 Traffic generator

The traffic generator module should be programmable with respect to:

- Origin: (a) single client ID or (b) random clients;
- Target: (a) single theatre or (b) random theatres;
- Operation: (a) query or (b) purchase;
- Rate: [req/sec];
- Duration: [sec].

In “random clients” mode, the client id is set randomly in the range 1-NrCl. In “random theatres” mode, requests are distributed randomly to theatres in the range 1-NrTh. In “query” mode, the generator just issues a search and then cancels. In “purchase” mode, the transaction is followed all the way through to the end. Rate of request bounds should be tested in the actual system.

You should be able to **launch at least two traffic generators in parallel**, e.g. one to generate background load (random), one or more to generate traffic to targeted theatres.

To measure the system performance, the user/tester should be able to see at any time:

- the active tests and their parameters;
- the number of discarded requests (replied with a ***<busy>*** exception);
- the percentage of discarded requests;
- the actual average rate at which each individual APP or DB server is serving requests;
- the average request execution latency.

3.2.2 Failure generator

The failure generator module induces or simulates failures of individual servers, either APP or DB. Likewise, it must also induce the restart thereof.

To be aware of the system health, the user/tester should be able to see at any time the state of the servers (up, down).

3.3 Tools and environment

A competent designer should choose the programming and runtime support tools he/she deems adequate to achieve the objectives, in the most optimized way.

However, the objectives of the work include mastering internal design details that may already be supplied by existing tools, some even discussed in this course. As such, we give guidelines about the architecture and which tools to use:

- Server-side software includes several main modules to be considered:
  - WEB service, to interface the clients;
  - application service APP with the WideBox logic;
  - database service DB to update, store and render the state of theatres (namely, availability of seats);
  - middleware to help design and implement the functional and non-functional attributes of the application and/or the database manager.
- We are looking at an architecture featuring:
  - WEB server in a separate machine, the client front-end;
  - the WEB server launches requests to the APP server;
  - the APP server reads and updates the DB server which keeps the state of the theatres;
  - The WEB server must be in a separate machine than the others, since in such a large scale system, having load balancing of HTML requests to web server pools is the least option for scalability;
  - The web pool will be simulated by traffic generation directed onto the APP server. In consequence, the traffic generator pretends to be (i.e., plays the role of) a set of web servers sending traffic directly to the APP sub-system, in the same syntax/semantics as, and in parallel with, the web server requests from the human-generated traffic (from client web browser(s)).
  - APP and DB server may co-reside, but they will potentially be in different machines, as the project phases unwind and performance and dependability requirements are addressed, e.g. with replication.
- The servers will be implemented in the DI lab machines, in the student areas.
- The clients can be any machine connected to the lab network with a web browser.
- Web server will be implemented on Apache[1] or other open-source SW.
- Application (APP) and database (DB) servers should be kept as simple as possible, and be implemented by the students.
- Use of ready-made DBMS is forbidden, students should implement the DB natively over files, aim at a key-value NoQSL storage, and guarantee persistence of write operations, by forcing timely **synchronous writes** into the disc, or alternative measures achieving the same effect.
- Use of facilitating coordination middleware for meeting the performance and dependability requirements is allowed, like for example, ZooKeeper[2].

[1] http://en.wikipedia.org/wiki/Apache_web_server
[2] http://zookeeper.apache.org/

- Traffic generator could be implemented as a stand-alone web-oriented tool but, for ease of implementation, it should be implemented on the web server side. This way, the tool will communicate with the APP server using exactly the same kind of mechanisms, syntax and semantics that will be developed for the submission of actual requests coming from the web. Use of metering software is allowed, as long as it meets all requirements for the traffic generation (e.g., Jmeter[3]).
- Programming language will be Java in general. Use of other languages for specific points may be considered, if justified.
- Client-side and server-side software can be designed at the students' choice (applets, JavaScript, CGI, servlets, etc.).
- Java RMI[4,5] is advised for programming the distribution aspects of the project, like for example the remote client-server invocation from web to APP server. Invocations should be non-blocking, for performance reasons.

4 Design hints

- The interface seen by the client may be plain text. E.g., a list of available seats, a list of occupied seats, and the reserved seat. A bit fancier, but **not required**, would be displaying places in columnar format, with colour coding for marking **free**, **occupied** and **reserved** seats (in reality, other customers' *reserved* places should be seen by each customer as *occupied,* but it is handier for us to see the room state this way). E.g. in format:

  A01 A02 A03 A04 ..... A38 A39 A40

  B01 B02 B03 B04 ..... B38 B39 B40

  .....

  Z01 Z02 Z03 Z04 ..... Z38 Z39 Z40

- Likewise, the interface given to the client to enter requests DOES NOT need to be mouse or cursor addressing-oriented, it may be plain text, for example:

  Please enter one of: YES for accept this seat; CAN for cancelling; XYY for choosing another seat: _____

  A sophisticated graphics user interface is not required, no penalty for that. However, it may be incrementally appreciated but only in the top tier of marks for the project.

- APP and DB server may co-reside in the initial phase, but they will potentially be in different machines, as the project phases unwind and performance and dependability requirements are served, so you might consider having them separate right from the beginning.
- A tool like **ZooKeeper** can help on several things: coordination of accesses to the several servers potentially involved in the design; management of the

[3] http://en.wikipedia.org/wiki/Apache_JMeter , http://jmeter.apache.org/
[4] http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html
[5] http://en.wikipedia.org/wiki/Java_remote_method_invocation

configuration of a fragmented (sharded) and/or replicated key-value storage; management of the consistency of any replicated entities for fault tolerance.

- To keep it simple, for best performance, database and database manager should follow the NoSQL and key-value store paradigm, similarly to Cassandra[6].
- To keep it reliable for consistency, don't forget that writes (e.g., committing a ticket purchase) must be **synchronous**. Otherwise, persistence guarantees are lost.
- To address the performance requirements, you may consider fragmenting and distributing (**sharding**) the DB organization, by splitting the DB through several servers to gain parallelism and thus performance, but remember that a no-sharing structure facilitates sharding in a way to gain performance[7].
- To achieve fault tolerance (in phase III), the use of replication is suggested. However, the use of replication in any of the phases is allowed, namely since it can also improve performance, through parallelism. Remember that **performance is a grading criterion**!
- To meet the dependability requirements, consider what is the most efficient way to handle the potential failure of APP or DB server, e.g. by combinations of replica and fragment placing, taking into account the eventual measures you already took to address the performance requirements.
- To perform metrics (traffic and failure), make it as easy as possible, forget aesthetics, we want semantics.
- For the traffic generator algorithm, think about what is the best way to generate the highest possible rate of **effective** seat purchases (given the Functional Requirements Specification); implement that for a single theatre and test; now modify the module to simulate random requests for different random theatres. **The use of multi-threading is recommended**.
- For the traffic generator interface, since the tool is implemented as a standalone application that directly connects to the APP layer, an elaborated tool interface is not required. What is really important is to make it easy to setting the parameters, to repeat past tests without re-entering parameters (e.g. creating pre-sets), and to observe the collected metrics.
- For the failure generator algorithm, consider that it can be implemented manually by stopping/restarting a server in the command line, in each local machine. Alternatively, a generic tool can be devised to inject failures/restarts remotely in any server and keep a status thereof (e.g., implementing specific methods in the server to make it stop replying to requests and start discarding them silently, and to make it resume serving requests, possibly after resetting its state). Use of Zookeeper to coordinate stopping and restarting of servers is also a possibility.
- For the failure generator interface, an elaborated tool interface is not required. What is mostly values is that stopping/restarting of servers is done easily and also that it is possible to see, at any moment, which servers are alive.

[6] http://en.wikipedia.org/wiki/NoSQL , http://en.wikipedia.org/wiki/Apache_Cassandra

[7] http://dbshards.com/dbshards/database-sharding-white-paper/ , http://en.wikipedia.org/wiki/Sharding

5 Delivery

5.1 Project phases

The project will be developed in three phases. They serve as milestones that help the student to keep pace and fulfil the main objective. The phases are not independent, so failure of a phase will condition the following phases. On the other hand, failure of later phases still allows a partial grade to be obtained from previous phases. The objectives of each of the three phases are the following:

I. **Design and implementation of the full application logics**, obtaining a fully functional system according to the requirements, without concern for non-functional requirements except that the system must ensure consistency of individual transactions, and work well under moderate load. **Design and implementation of the traffic and failure generators** according to the requirements. A differentiating grading parameter besides the above will be the maximum rate of served requests by the WideBox service before saturation.

II. **Demonstration** of the capability of withstanding very large-scale **continued workloads**, for **all the theatres, AND** very large-scale **peak workloads** for **specific theatres**, according to the requirements, by **designing and implementing** the necessary mechanisms to enhance the initial design. A differentiating grading parameter besides the above will be the maximum rate of served requests by the WideBox service before saturation in either case.

III. **Demonstration** of the capability of withstanding **failure of any single individual server** and remaining available, both under normal and under heavy load conditions (APP or DB) by **designing and implementing** the necessary mechanisms to enhance the previous design. A differentiating grading parameter besides the above will be the stability of served requests rate through failure.

5.2 Project demonstration

Students should be prepared to demonstrate any of the functionality requested and/or developed. At the very least, in order to demonstrate the project capabilities listed in the previous section, they should be prepared to swiftly demonstrate:

I. Regular manual access from client browser.
II. Same as I, with background random traffic.
III. Same as II, with heavy background random traffic up to saturation.
IV. Same as I, with peak traffic to a single theatre.
V. Same as II, simulate single DB unit failure.
VI. Keep V, now simulate single APP unit failure.
VII. Keep VI, simulate global DB units failure.

5.3 Submission requirements

In each phase of the project, each group must submit a report and the project code, and:

- All materials are submitted through the course web page in Moodle.
- The report **cannot exceed 6 pages in 12pt font.**
- All materials (developed code and report) must be wrapped (zip, rar, 7z, etc.) into a **single file.**

- **All files (code and report) must be identified with the group number and group members.**
- The archive file must be named as PSD1617-Project-PhaseXX-GroupYY.(zip, rar, etc.)

5.4 Deadlines

- Phase I – October 21st, 2016, until 23h55.
- Phase II – November 16th, 2016, until 23h55.
- Phase III – December 9th, 2016, until 23h55.