Distributed Systems (CS236351)

Winter, 2013-2014

Exercise 3

Due date: 30/12/13, 23:59

1 System overview

In this exercise, you are going to develop another version of the basic flight search service, which you implemented in the first assignment. Read the assignment carefully, as the requirements from all considered components have been modified.

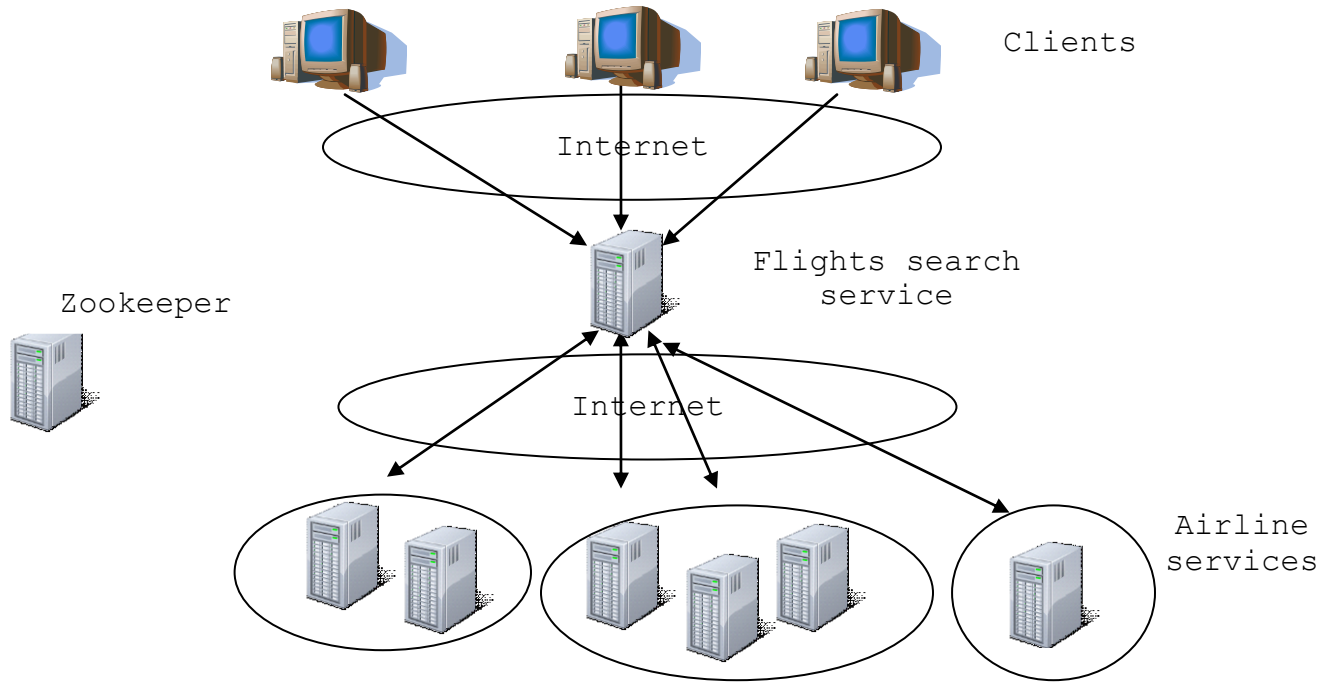
The service will allow clients to submit a query for a flight after specifying several parameters, such as the source and destination and the requested date. During query processing, it will contact airline companies, which were registered to the service, and return results received from these companies to the client.

The service will support just one operation:

`Search` – searches for flight routes between the given source and destination airports, which leave at the given date (for simplicity, the service will handle one-way flights only). The response of the service is a list of records containing the information on the proposed flight routes and airlines serving each of the proposed flights.

2 System components

The system components are described in the figure below and elaborated in the following.



2.1 Client

The system supports several clients. Each client is an interactive console application that communicates with the flights search server over the Internet. The console application supports the following commands:

- `search <src> <dst> <date> [<Airline server1>] [<Airline server2] … [<Airline serverX>]` – searches for flights leaving from the specified source to the specified destination. If Airline servers were specified, searches only for flights of those airline servers. If no Airline servers were specified, searches all Airline servers. The response of the flights search service includes information on the proposed routes.
  For example:

```
> search tlv jfk 10/11/2010
400$: tlv-par (air-france, af1234), par-jfk (air-france, af321)
500$: tlv-jfk (delta, dl547)
700$: tlv-fra (lufthansa, lf558), fra-jfk (united, un9875)
1200$: tlv-jfk (elal, ly6637)
> search tlv jfk 10/11/2010 air-france delta united
400$: tlv-par (air-france, af1234), par-jfk (air-france, af321)
500$: tlv-jfk (delta, dl547)
```

  The results should be sorted according to the total cost of the suggested routes
- `exit` – closes the client application.

2.2 Airline server

The airline servers are grouped into clusters representing airline alliances. For example, "SkyTeam" cluster consists of servers of Alitalia, AirFrance, etc., while "StarAlliance" cluster consists of servers of Lufthansa, Swiss, etc. Every cluster must appoint a **delegate** Airline server that will receive and handle all queries from the flight search server. The **delegate** server should forward all queries to the appropriate Airline servers.

When a server A belonging to a cluster X is searched for a flight, it should contact all (appropriate) other servers belonging to the cluster X to check whether they have a connection flight from one of the destinations served by A. The connection flight must depart at the same date or the day after the first part of the route is taken. For example, if A has a flight from TLV to NYC at 10/11/2010 and B (which belongs to A's cluster) has a flight from NYC to TOR at 11/11/2010, A has to return the route TLV-NYC-TOR in response to the query for flights between TLV and TOR at 10/11/2010. For simplicity, we will assume all routes might have at most one connection stop. All communication between servers belonging to the same cluster should be done using web services. All returned flight must be from the Airline servers specified in the query (if any were specified).

In order to provide fault tolerance, you have to implement a replication mechanism, which will ensure that the data of each of the airline servers is stored on some other server(s) belonging to the same cluster. For example, given a cluster consisting of three servers A, B and C, if A fails, its data on available flights should be managed by B and/or C. Your replication algorithm should provide load balancing among all servers in the cluster, and should be resilient to up to `n-1` failures for clusters consisting of `n` servers. Notice that failed servers can be restored later with new flights data. In such a case, the old replicated data should be discarded, and the new data should be replicated.

The airline servers should expose methods to query for available flights. You are free to choose the architecture of the service (REST or SOAP). Upon the initialization (or failure recovery), each server will be provided with a text file containing information on available flights. The file will contain lines in the following format:

```
<flight number> <src> <dst> <date> <price>
```

Notice that the data file may change when the failed server is restarted.

2.3 Flights search server

The flight search server should expose methods to support clients' queries. In addition, it should expose methods for the addition/removal of airline servers.

Each search query should be forwarded to the appropriate **delegate** airline servers available at the time of the request. A recovery mechanism should take care of **delegate** airline servers' failures, and produce a valid response to the client even if one or more such failures have occurred. You may assume that the flights search server is always up and available.

3. ZooKeeper

A server that needs to use ZooKeeper service should have a special configuration file present in the run directory of the servers' executable. The configuration file should be named "zookeeper.conf". It should contain a single line: a comma separated list of host:port pairs, each corresponding to a ZooKeeper server.
For example:
"127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002"

4. WCF Extensions

You should implement the following mechanisms through WCF extensions:

- **Caching** – utilizing the fact that the data is (almost) static, efficient caching of query results is possible. You need to design and implement the caching mechanism in the system at all appropriate components, storing the latest 100 query results and retrieving them when appropriate. Notice that caches might become invalid in case of an airline server failure and/or recovery.
- **Logging** – each request to the flights search server should be logged. Each log should include information on the parameters of the query and the duration of the service. The

name of the log file will be given as a command line parameter to the flights search server.

5. Dry question

As you can see, the requested functionality has been reduced to only search queries that do not change the flight databases.
Describe how you would implement reserve/cancel queries.
If you can think of several possible implementations – describe them and their tradeoffs.
Describe any changes you would make to the replication mechanism for each implementation (if any).
You should answer this question as a part of the documentation.

Assumptions

While solving the exercise, you may assume the following:

- Airline servers may come up and fail at arbitrary times under following restrictions:
    - After each failure/addition of an airline server in cluster X, there is some (short) grace-period during which the group of airline servers in cluster X does not change.
        - **Hint**: use this assumption while designing your replication mechanism.
        - **Hint2**: on addition of an airline server, the grace period is not long enough to send replicas to the entire cluster.
    - Changes in different clusters may occur concurrently.
    - During the changes in clusters, a delegate airline server might return an incomplete answer. In particular, a delegate airline server should not wait until the replication of the data of new/failed airline servers is complete.
    - You should consider in your solution that the delegate airline server may fail like any other airline server. Ensure that the cluster is able to recover from delegate airline server failure.
- All components are provided with valid command line arguments, as shown below. In particular, input files for the airline servers are in the specified format.
- You may not store all flight data on the ZooKeeper servers. Assume that the data can be too large.

Additional details

- All communication should be made with WCF or using ZooKeeper. Though you may deploy all the components as processes on a single machine, you should consider the specified architecture.
- The flight search server should query only the delegate airline servers.
- The clients should not "see" the description of the interface between the flight search server and airline servers.
- Airline servers may fail at arbitrary times, including during the processing requests from some client(s). Clients should be unaware of these failures.
- There might be clusters with just one airline server. In this case, the replication is not required.
- Pay special attention to **load-balancing, efficiency** and **fault-tolerance** aspects of the system.
- **It is up to you to complete the design and protocol details of the system**, in particular, the replication protocol, the caching mechanism, the architecture of the airline servers and the required data contracts. **You need to describe and justify any design decision you take in the documentation.**
- Use **exception handling mechanisms** to report on internal errors in flight search service and airline servers.
- Make your console applications "user friendly". Print messages to show progress and feedback to the user.
- Your code should be reasonably documented and understandable.
- A detailed external documentation should describe how you solved the exercise and, in particular, explain and argue your design choices. Please, take the external documentation seriously**, it will consist a substantial part in grading your work**. Note that you may lose points if some design issues are not described, or there are inconsistencies between the source code and the documentation. **Pay special attention to the replication protocol,** explain how it works, how it keeps the load balanced and also its efficiency (for example, how much messages are sent to ZooKeeper servers for every operation)
- Invest considerable time in testing your application.
- An input file example for the airline servers is published with the exercise. Use your own, more sophisticated input files to test your application.

Submission

Submission is in pairs only and using the electronic submission system. The submitted filename should be ds-ex3.zip

The file should contain:

1. A text file named submitters.txt containing the names, IDs and emails of the submitters.
2. Another zip file with your entire Visual Studio solution.
3. A doc/pdf file with your external documentation.
4. A zipped folder named `run`. The folder should contain 3 executables named (with respect to their roles) `Client.exe`, `FlightsSearchServer.exe`, `AirlineServer.exe`. They will all be invoked from the `run` folder using the following syntax:

Client.exe <flights search server URI #1>

e.g., **client.exe localhost:8000/Services/FlightsSearch**

FlightsSearchServer.exe <clients port> <sellers port> <log file>

e.g., **FlightsSearchServer.exe 8000 8100 log.txt**

AirlineServer.exe <name> <alliance> <search server port> <airline servers port> <flights search server URI #2> <input file>

e.g., **AirlineServer.exe KLM SkyTeam 8200 8201 localhost:8100/Services/FlightsSearchRega.txt**

<search server port> is the port for search queries.
<airline servers port> is a port for intra-cluster communication.

Also, add a text file named **commands** including examples of running each of the executables.
(In particular, correct < flights search server URI #1> and < flights search server URI #2>)
Add any other required files and make sure all executables can be run from the submitted directory.
Points will be deducted if the executables could not be run from the run directory.

Good luck!