Team 4: Unicommerce

17-654: Analysis of Software Artifacts
18-841: Dependability Analysis of Middleware

Wei Zhang
Dawei Gu
Maneesh Sharma
Morgan Linton

Team Members

Wei Zhang
wzhang@cs.cmu.edu

Dawei Gu
dwgu@cs.cmu.edu

Maneesh Sharma
maneeshs@andrew.cmu.edu

Morgan Linton
mlinton@andrew.cmu.edu

http://www.ece.cmu.edu/~ece841/team4
Baseline Application

- A commerce application geared towards college students buying and selling goods.
- The baseline application allows users to buy, sell, search, and execute a cancel sell operation as well as browse usage history.
- This is a very interesting application because it provides a great advantage over CMU’s Misc. Market. The main features of this application are:
  - items are stored in a database
  - items are removed from the database when sold
  - users can view usage history
  - users can search for items meeting a specific criteria.

Middleware Platform: CORBA
- Chose to use CORBA because all members of the group were unfamiliar with it and excited by the ubiquity of this middleware platform. We thought that the popularity of this platform would make it easier to get help and access online resources.

Language: Java (J2SE v. 1.4.1)
- All team members were very familiar with Java. This meant that we would not have to worry about a team member having to learn anything on top of CORBA.

3rd Party Software: mySQL (used initially)
Microsoft SQL Server (used in final implementation)
Jbuilder
ORBD
- We switched from mySQL to MS SQL Server so that we could have stronger server-side scripting capabilities.
Fault-Tolerance Goals

- Fault-tolerance goals
  - We are simply replicating the server four times in our system so that if one server goes down, there will be another one waiting to accept client requests.
  - All state in our system is stored in the database.
  - Each server can be on a different machine.
  - Database and Replication Manager are stored on the Sacred Machine
Fault-Tolerance Goals

- Elements of fault-tolerant framework?
  - Replication Manager
  - Fault detector
  - Automatic recovery (for four servers)
  - Fault injector

Fault-Tolerance Strategy

- Warm Passive Replication
- Replication Manager – New component
  - Periodical Fault Detection
  - Service-Rebind Failover
  - Automatic Fail Recovery
- Server Replica – New CORBA interface introduced
  - Responds to the Replication Manager’s instructions
  - Registers itself as a primary replica when instructed
  - Provides service to Clients
- Client – No change
  - Failover is completely transparent to the clients
  - Clients always request service using the same object name
Fault-Tolerance Architecture

Fail-Over Measurements

Fault-Tolerant Round Trip Times

Legend: Machines Client App Server Component DB
Fault-Tolerance Analysis

- Fault Detection Polling Rate
  - Every $n$ ms the replication manager checks to see if the servers are alive.
  - Decreasing the polling period allows the Replication Manager to recognize dead servers sooner.
  - To reduce the fail-over time we decreased the polling period for the replication manager.

Polling Rate vs. Average Process Time
FT-RT-HP Strategy

- Address new issues and balance tradeoffs
  - Fault-Tolerance: Continue to use replicas, no rebind
  - Real-Time: Properly lower the fault detect polling rate
  - High-Performance: Use load balance

- Introduce Factory
  - Consistent with other components by using CORBA solutions
  - Cross different platforms
  - Hide physical location of replicas from Replication Manager

- Replication Manager
  - (also) Acts as Fault-Detector and Load Balancer
  - Detects whether the replicas are still alive
  - Launch new replica through Factories if the old one is dead
  - Direct the Clients to less loaded Replicas

- Both the Server Replica and the Client do not change
  - All procedures are transparent to the Clients the Server Replicas
  - Client always requests service based on the name given by Load Balancer

FT-RT-HP Architecture

[Diagram showing the architecture of FT-RT-HP system]
Real-Time Measurement

Overall Operation Time (ms)

<table>
<thead>
<tr>
<th></th>
<th>50 ms</th>
<th>800 ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst Case</td>
<td>3570 ms</td>
<td>4320 ms</td>
</tr>
</tbody>
</table>

Time breakdown

Fault-Free Time breakdown

Fail-over Time Breakdown
High-Performance Measurements

![Graph showing overall operation time for 1 and 3 replicas]

- 1 Replica Average: 283ms
- 3 Replicas Average: 48ms

Other Features

- Extra Features:
  - Duplicate detection for database entries
  - Graphical Interface for client, server, and replication manager
  - Automated fault injection system
    - Number of Faults to Inject
    - Fault Interval (ms)
    - ORB Host Address, ORB Port
  - Automated testing system
    - Number of Clients
    - Test Number
    - Interval (ms)
Lessons Learned

- How to analyze a project from a high-level and break it down into discrete realizable components.
- How to determine the information needed for testing situations in incorporating that into an automated system.
- What details were required to build an effective fault injection system that would work hand-in-hand with our automated testing architecture.

Insights from Measurements

- Fault-tolerant
  - Fail-overs produce large spikes. Our hypothesis is that these are due largely to the fault-polling interval.

- Real-time
  - The higher the polling frequency, the faster the overall response time.

- High-Performance
  - Load balancing provides a faster overall operation time than directing all traffic to a single server.
Open Issues

- Additional Features that we would have liked to have implemented in our system:
  - More sophisticated load balancing heuristic
  - Emailing sellers when their items are purchased
  - Credit-card transaction system

Conclusions

- What we learned in this project:
  - In this project we learned that there is no feasible methodology that can be used to create an application that optimizes fault-tolerance, real-time, and high-performance.
  - For real-world systems we discovered that one can optimize the parameters that are most crucial to the system while allowing for some degradation of service in a particular area of less importance.
Conclusions

- If we could start the project from scratch we would do the following things differently.
  - Consider fault-tolerance at the beginning of the project
  - Put more effort into identifying state
  - Try active replication and caching