High Performance CORBA

Shikharesh Majumdar

Real Time and Distributed Systems Lab
Dept. of Systems & Computer Engineering,
Carleton University,
Ottawa, CANADA.
email: majumdar@sce.carleton.ca
ACKNOWLEDGMENTS

COLLABORATORS:

» Istabrak Abdul-Fatah
» Imran Ahmad
» Siva Balasubramaniam
» E-Kai Shen
» Yiping Shi
» E-Kai Shen
» Wai Keung Wu

SPONSORS:

» Natural Sciences & Engineering Research Council of Canada (NSERC)
» Communications and Information Technology Ontario (CITO)
» Nortel Networks
» Department of National Defense (CANADA)
Part I: Introduction

Part II: Impact of Interaction Architectures

Part III: Performance Optimizations for Systems with Limited Heterogeneity

Part IV: Application Level Design Guidelines for Performance Enhancement

Part V: Real-Time Issues
PART I: INTRODUCTION

Heterogeneity in DOC Systems

- Distributed Object Computing (DOC)
  - Concurrency, Reliability
  - Reuse

- Heterogeneity: inherent in DOC
  - System upgrade
  - Addition of new components

- Middleware provides inter-operability in heterogeneous environments
• Both clients and servers use a standard API
• Client binds to server object with the help of agent
• Client invokes method in server
• Server receives request, performs requested operation, and sends reply
A standard put forward by OMG - a consortium of a large no. of companies

Basic idea: to access services provided by a server object through a *pre-defined* interface

IDL: Interface Definition Language

Stubs & Skeletons: provide marshalling/unmarshalling of data

CDR: Common Data Representation

GIOP: A protocol for inter-ORB communication (IIOP is GIOP over TCP/IP)
  » IIOP is GIOP over TCP/IP
Scalability is a desirable attribute for most systems

Many real-time applications also require low latency and high bandwidth
  » Telephone Switch
  » Embedded Systems

A drawback of most Commercial-Off-The-Shelf CORBA compliant ORB products:
  » often sacrifices performance for providing inter-operability

**Research Goal:**
Develop low overhead high performance middleware systems by
  » Using innovative client-agent-server interaction architecture *(Part II)*
  » Exploiting limited heterogeneity in systems *(Part III)*
  » Using application level performance optimizations *(Part IV)*
PART II: INTERACTION ARCHITECTURES

- Goals
- The H-ORB Architecture
- The F-ORB Architecture
- The P-ORB Architecture
- Performance Comparison of Interaction Architectures
- Summary of Observations
- The Adaptive ORB
GOALS

- Understand the performance impact of client-agent-server interaction architecture on performance
- Compare different interaction architectures
- Gain insight into scalability and latency of these architectures and understand their relationship with workload
- Devise an interaction architecture that produces high performance under a variety of different workload conditions
THE HANDLE DRIVEN ORB (H-ORB)
Fig. 4.8 from [Abdul-Fatah]
Use a commercial CORBA compliant ORB: ORBeline (Visigenic Systems)

Conduct experiments on a network of workstations running Solaris 2.5

**Synthetic application:**

- **Client:** operates cyclically
- **Server:**
  - Two copies of each server Si (i=1 or 2)
  - When invoked a server instance executes a for loop to consume a pre-determined amount of CPU time.
DESIGN OF EXPERIMENTS

- **Primary Factor:** \( N \) (no. of clients)
  - Used to control system load

- **Secondary Factors:**
  - Message Size \( (L) \)
  - Server Demands \( (S_1, S_2) \)
  - Inter-node Delay \( (D) \)

- Relative performance of the three architectures observed to depend on these factors.

**Performance Measures:**
- Throughput (client requests/sec)
- End-to-end response time
- Utilization (Process, CPU)
EXPERIMENTAL RESULTS: SOFTWARE BOTTLENECKS

- The Agent saturates and limits performance (*software bottleneck*)
- Synchronous communication:
  - agent waits until message is received

Fig. 5.10 and 5.12 from [Abdul-Fatah]
EXPERIMENTAL RESULTS: PROCESS CLONING

Fig. 5.15 and 5.19 from [Abdul-Fatah]

- Degree of Cloning = 4
- Degree of Cloning = 8
EXPERIMENTAL RESULTS:
SHORT MESSAGES

Fig. 5.23 and 5.33
- No Cloning
- Degree of Cloning = 4
- Higher Throughput
EXPERIMENTAL RESULTS:
LONG INTER-NODE DELAYS

Fig. 5.37 and 5.42

- No Cloning
- Degree of cloning = 4
- Higher Throughput
SUMMARY OF OBSERVATIONS

- Possible to change interaction architecture of a COTS ORB
- Each architecture has a “winning” attribute
  » H-ORB: Light weight in construction
  » F-ORB: Uses 3 messages per interaction
  » P-ORB: Invokes servers concurrently

- Latency-Scalability Tradeoff
- Agents can become software bottlenecks
  » Solution: agent cloning
- P-ORB: Tradeoff between concurrency of execution and waiting
- Communication-Bound Systems: H-ORB is a better choice as long as message transfer delays are not very high
- Computation-Bound Systems: F-ORB AND P-ORB seem to be preferable
ADAPTIVE ORB

Similar to Fig. 3.3 from [Shen]
Fig. 3.6 from [Shen]
EXPERIMENTAL ENVIRONMENTS AND EXPERIMENTS

- Experimental Environment:
  - Network of Sun workstations running under Solaris
  - Middleware: Orbix

- Primary Factors
  - Threshold QL:
    - if length of forwarding queue > QL then return server handle to client
    - else forward message to server
  - QL=24: Forwarding ORB; QL = 0; Handle Driven ORB
  - N (no. of clients) -- Used to control system load

- Secondary Factors:
  - Message Size (L)
  - Server Demands (S1, S2)
  - Inter-node Delay (D)
ADAPTIVE ORB: RESULTS (EFFECT OF MESSAGE LENGTH)

Fig. 5.15 and 5.17 from [Shen]

- \( L = 4800 \text{ Bytes}, \)  
- Adaptive ORB performs the best

- \( L = 19200 \text{ Bytes} \)  
- Adaptive ORB produces the highest performance
Fig. 5.21 and Fig. 5.23 from [Shen]

- D = 50 ms,
- Adaptive ORB performs the best

- D = 250 ms
- Adaptive ORB produces the highest performance
Fig. 5.33 from [Shen]
- Server Closer to Agent
- Adaptive performs the best

- Similar to Fig. 5-35 [Shen]
- Client Closer to Agent
- Adaptive ORB produces the highest performance
Polling Interval = 1 ms

- Increasing the degree of multithreading increases throughput

- Increasing the degree of multithreading further decreases throughput
ADAPTIVE ORB: NEW RESULTS
(EFFECT OF MULTITHREADING)

- Polling Interval = 100 ms

---

Fig. 5.43 and Fig. 5.44 from [Shen]

- Throughput increases monotonically with the degree of multithreading
How to determine threshold QL?
  » Use Analytic/simulation model
  » tuning parameter

Based on principles of balancing the load between forwarding and returning queues

if \( Q_f < Q_r + 1 \) {
  insert request in forwarding queue

} else {insert request in returning queue
  }

Dynamic threshold (load balancing) performs better than using any static QL value

### Throughput (\( N = 8, L = 150 \) bytes, \( SA, SB = 10, 15 \) ms, \( D = 50 \) ms)

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4.4997</td>
</tr>
<tr>
<td>1</td>
<td>8.8298</td>
</tr>
<tr>
<td>2</td>
<td>8.9266</td>
</tr>
<tr>
<td>3</td>
<td>8.0328</td>
</tr>
<tr>
<td>4</td>
<td>7.1868</td>
</tr>
<tr>
<td>5</td>
<td>6.2472</td>
</tr>
<tr>
<td>6</td>
<td>5.8592</td>
</tr>
<tr>
<td>7</td>
<td>5.8399</td>
</tr>
<tr>
<td>8</td>
<td>5.8460</td>
</tr>
<tr>
<td>24</td>
<td>5.8502</td>
</tr>
<tr>
<td>Dynamic</td>
<td>8.9802</td>
</tr>
</tbody>
</table>

Dynamic Threshold produces best performance (preliminary results)
Adaptive ORB combines the good attributes of H-ORB and F-ORB
- Highest performance under many workload conditions
- up to 100% improvement in performance observed for certain workload parameters
- Close to highest performer for other workloads
- Multithreading can be used to enhance performance at higher loads

Determination of Threshold
- Simulation and analytic models
- Tuning parameter

Dynamic Threshold-based technique looks promising
PART III: SYSTEMS WITH LIMITED HETEROGENEITY

- Certain systems may be characterized by limited heterogeneity
  » embedded systems: most system components may be built using the same programming language and platform.

- “Fly over” the CORBA protocol when two similar components talk to each other

- **Goal**: Investigate optimization techniques that exploit limited heterogeneity
POSTMORTEM OF A TYPICAL CORBA METHOD INVOKATION

Fig. 4.1 from [Ahmad]
RESULTS OF POSTMORTEM: DISTRIBUTION OF WORK

- **Experimental Setup:**
  - Client and server execute on a network of PC’s running under Linux
  - Middleware: ORBACUS
  - A simple server that inserts a value into a variable
  - marshalling time (M)
  - unmarshalling time (U)
  - communication time (C)
  - server execution time (S)

**Proportion of End-to-End Response Time**

Case 1 - Data type: Octet

Case 2 - Data Type: Double
OPTIMIZATION STRATEGY 1: PREVENTION OF “NATIVE - NEUTRAL” DATA CONVERSIONS

- **Marshalling**: Convert data in *native* format into data in a *neutral* format.
- **Unmarshalling**: Convert data in *neutral* format back into *native* format.
- Data conversion burns CPU cycles.

- **Optimization**: If client and server are NOT heterogeneous
  - DO NOT perform data conversions

(achieved through modification of ORB source code)
OPTIMIZATION STRATEGY I: RESULTS

- **Experimental Setup:**
  - Network of PC’s running under Linux
  - Middleware: ORBACUS
  - A simple server that inserts a value into a variable

- **Sources of savings:**
  - Reduction in computation time (no data conversions) and data copying

Fig. 5.2 from [Ahmad]
OPTIMIZATION STRATEGY II: REMOVING PADDING IN CDR

- CDR aligns primitive data types on byte boundaries that are natural for most machine architectures.

```c
struct CD {
    char c;
    double d;
};
```

- Remove padding before transmission of message and re-introduce padding after reception of message
  - achieved through interceptor programming
OPTIMIZATION STRATEGY II: RESULTS

- Padding Removal: communication-computation tradeoff
  - Savings: communication times
  - Overheads: increase in marshalling/unmarshalling times
- Performance improvement is possible for communication intensive applications

- **Experimental Set up:** same as before
- Zone A: Loss in performance   Zone B: Gain in performance

*Fig. 5.3(b) from [Ahmad]*
OPTIMIZATION STRATEGY II: RESULTS (CONTD.)

- Significant benefit is possible for data intensive applications
- Improvement in performance is proportional to the percentage of padding bytes in the message

Net savings: savings in communication time - increase in marshalling time

Fig. 5.3 (a) from [Ahmad]
Figure 3.6 from [Wu]

- CORBA ORB - when client and server are dissimilar
- Flyover - with similar client and server
Figure 3.5 from [Wu]

- Implemented in Perl
- Takes standard CORBA IDL file as input (transparency)
- Produces Client Stub and Serve Skeleton
- Choice of path at run time: default - flyover; client invokes a special method to use CORBA
References


Connection Set Up

- Location as well as state of servers and clients can affect the connection set up latency

Typical scenario:

1. Client wants to bind to server
2. Middleware looks up implementation repository
3. Middleware daemon activates server
4. Handle for requested server is returned to client
Factors that can affect system performance:

» State of server
  – Active (A) / Not Active (NA)

» Collocation of client and server in same address space
  – Collocated (C) / Not Collocated (NC)

» Distribution of client and server
  – Same Node (SN) / Different Nodes (DN)

Various combinations of these lead to different cases
### Different possible configurations

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Mode I Case I</th>
<th>Mode I Case II</th>
<th>Mode II Case III</th>
<th>Mode II Case IV</th>
<th>Mode III Case V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server state</td>
<td>NA</td>
<td>NA</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>Distribution</td>
<td>SN</td>
<td>DN</td>
<td>SN</td>
<td>DN</td>
<td>SN</td>
</tr>
<tr>
<td>Collocation</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
<td>C</td>
</tr>
<tr>
<td>Latency (ms)</td>
<td>171.8</td>
<td>174.4</td>
<td>54.6</td>
<td>55.2</td>
<td>0.9</td>
</tr>
</tbody>
</table>

#### Latency (ms)

![Latency Graph](image)
Server state has a large impact on connection set up latency
  - at least 300% improvement observed by keeping the server active
    - Keep at least the popular servers active

If client and server are on the same node collocation can improve performance significantly
  - an order of magnitude performance improvement is observed
    - Collocate clients and servers when ever possible

If client and server can not be collocated placing them on the same node can improve performance only marginally
  - performance improvement observed is of the order of 1%
    - migration of server from client node to a different node is unlikely to degrade connection setup latency
    - can balance load more evenly by distributing servers on different nodes
  [Note: higher network delays (WAN) may lead to a different conclusion]
Two issues
  » native - CDR conversion overhead
  » bandwidth savings

**CDR alignment of primitive fixed-length types [OMG-98]**

<table>
<thead>
<tr>
<th>Starting Bytes Boundary</th>
<th>IDL Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiples of 1</td>
<td>char, octet, boolean</td>
</tr>
<tr>
<td>Multiples of 2</td>
<td>short, unsigned short</td>
</tr>
<tr>
<td>Multiples of 4</td>
<td>long, unsigned long, float, enumerated types</td>
</tr>
<tr>
<td>Multiples of 8</td>
<td>long long, unsigned long long, double, long double</td>
</tr>
</tbody>
</table>

Example of complex data structure:

```c
struct CD {
    char c;
    double d;
}
```

Padding bytes in a stream
Complex Data Structure - Case A

- **Option I (Most natural)**
  
  » Method 1 with one parameter
  
  ```
  struct CD [size] {
      char c;
      double d
  }
  ```

- **Option II**
  
  » Method1 with two parameters
  
  ```
  char [size] c; double [size] d;
  ```

- **Option III**
  
  » Method1 with parameter char [size] c;
  
  » Method 2 with parameter double [size] d;
CASE A - RESULTS

<table>
<thead>
<tr>
<th>Array Size</th>
<th>1</th>
<th>100</th>
<th>1000</th>
<th>5000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Option 1 (ms)</td>
<td>2,131</td>
<td>3,472</td>
<td>24,556</td>
<td>123,703</td>
</tr>
<tr>
<td>Option 2 (ms)</td>
<td>2,014</td>
<td>2,314</td>
<td>8,356</td>
<td>41,489</td>
</tr>
<tr>
<td>Option 3 (ms)</td>
<td>3,747</td>
<td>4,298</td>
<td>15,490</td>
<td>74,168</td>
</tr>
</tbody>
</table>

- parameter passing has a large impact on performance (increases with size)
- separating data types leads to large performance savings (less padding bytes)
- multiple method invocations may be preferable to single method invocation with complex parameter
Complex (Nested) Data Structure - Case B

- Option I (Most natural): Method 1 with one parameter
  ```c
  struct CSB [size] { 
    char [size] c;
    struct s { int x; int y; int z;
    }
    boolean b;
  }
  ```

- Option II: Method 1 with three parameters
  ```c
  char [size] c;
  struct s {int x; int y; int z;}
  boolean [size] b;
  ```

- Option III: Method 1 with 5 parameters: char [size] c; x; y; z; boolean [size] b;

- Option IV:
  Method1 with parameter char [size] c; Method 2 with parameter x;
  Method 3 with parameter y; Method 4 with parameter z; Method 5 with parameter b.
CASE B - RESULTS

Response Time of Case B and the Comparison of Four Different Options

<table>
<thead>
<tr>
<th>Array Size</th>
<th>1</th>
<th>100</th>
<th>1000</th>
<th>5000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case B.1 (ms)</td>
<td>370</td>
<td>384</td>
<td>419</td>
<td>690</td>
</tr>
<tr>
<td>Case B.2 (ms)</td>
<td>362</td>
<td>375</td>
<td>502</td>
<td>560</td>
</tr>
<tr>
<td>Case B.3 (ms)</td>
<td>360</td>
<td>363</td>
<td>388</td>
<td>511</td>
</tr>
<tr>
<td>Case B.4 (ms)</td>
<td>976</td>
<td>980</td>
<td>1011</td>
<td>1321</td>
</tr>
</tbody>
</table>

- Unwrapping parameters improves performance: largest performance improvement occurs when all primitive types are separated
- Invoking multiple methods produces worst performance
PARAMETER PASSING - OBSERVATIONS

- Parameter passing strategy is important: large performance benefits can accrue from adopting an appropriate parameter passing mechanism

- Two types of overheads:
  - data conversion (CPU operations in marshalling/unmarshalling)
  - padding bytes (communication bandwidth + CPU operations in marshalling/unmarshalling)

- Suggested practice: Unravel the complex data structures
  - Separating native data types into multiple arguments improves performance
  - [impact of padding byte removal seems to be the dominant factor]
  - Savings in term of padding bytes seems to be the most dominant factor

- When multiple methods are invoked successively
  - collapsing multiple methods into one may produce performance improvement (A.2 and A.3 & B.3 and B.4).
Experiment with one object and 500 methods.

- Measure response time for different method invocation

**Results of Experiment**

<table>
<thead>
<tr>
<th>nth-method</th>
<th>Latency (microsecond)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st method</td>
<td>16,406</td>
</tr>
<tr>
<td>100th method</td>
<td>16,742</td>
</tr>
<tr>
<td>200th method</td>
<td>17,210</td>
</tr>
<tr>
<td>300th method</td>
<td>17,537</td>
</tr>
<tr>
<td>400th method</td>
<td>18,019</td>
</tr>
<tr>
<td>500th method</td>
<td>18,352</td>
</tr>
</tbody>
</table>

Method dispatching: locate object and then locate method in object

- Seems to be a linear search (product dependent)
### Method Placement and Dispatching Latency

- **Single Interface**: all methods in one object (Model I)
- **Multiple Interfaces**: each method in a separate object (Model II)

#### Latency Measurement

<table>
<thead>
<tr>
<th>Method</th>
<th>Model I</th>
<th>Model II</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st method</td>
<td>16,445</td>
<td>17,051</td>
</tr>
<tr>
<td>10th method</td>
<td>16,496</td>
<td>17,065</td>
</tr>
<tr>
<td>20th method</td>
<td>16,542</td>
<td>17,092</td>
</tr>
<tr>
<td>30th method</td>
<td>16,573</td>
<td>17,096</td>
</tr>
<tr>
<td>40th method</td>
<td>16,639</td>
<td>17,118</td>
</tr>
<tr>
<td>50th method</td>
<td>16,689</td>
<td>17,124</td>
</tr>
</tbody>
</table>

- Single interface has a marginally better performance
  - packing multiple methods into one interface can give rise to performance improvement
PACKING OF METHODS & OBJECTS INTO SERVER

- How many methods should be packed in an object?
- How many objects should be placed in a server?

Mode I:
  - Pack all methods in a single object (server)

Mode II:
  - Pack each method in a separate object
  - Pack all objects in the same server
PACKING OF METHODS & OBJECTS INTO SERVER (contd.)

- **Mode III:**
  - One method per object
  - One object per server (same node)

- **Mode IV:**
  - One method per object
  - One object per server
  - One server per node
No. of Clients   | # 5  | # 8  | # 10 | # 15  
Mode I (Latency (ms)) | 100.0 | 152.8 | 190.0 | 276.9  
Mode II (Latency time) | 99.9  | 153.2 | 192.8 | 278.0  
Mode III (Latency time) | 96.8  | 155.0 | 219.4 | 433.1  
Mode IV (Latency time)   | 71.7  | 125.4 | 169.4 | 257.6  

![Latency (ms) Chart](chart.png)
Higher Service Time (10 ms) No. of Clients = 10

<table>
<thead>
<tr>
<th>Mode</th>
<th>Latency (ms)</th>
<th>Comparison (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode I</td>
<td>291.5</td>
<td>--</td>
</tr>
<tr>
<td>Mode II</td>
<td>291.8</td>
<td>+0.1%</td>
</tr>
<tr>
<td>Mode III</td>
<td>319.8</td>
<td>+ 9.71%</td>
</tr>
<tr>
<td>Mode IV</td>
<td>220.6</td>
<td>- 23.42%</td>
</tr>
</tbody>
</table>
Model IV (maximizing concurrency) produces best performance (expected)

Mode III (independent servers on same node) produces worst performance
   › process switching overhead

Mode I and Mode II perform comparably

Performance differences among modes increase with the no. of clients
Load Balancing by Application Designer

- Example system: Two classes of clients - busy and non-busy
- Replicated Servers
- Mode I: Unbalanced System
  - Two busy clients connected to the same server

Mode II: Balanced System
  - Load split evenly across two servers
LOAD BALANCING (Contd.)

- Mode III: Half Dynamic Self RR
  - Busy client selects server by using local RR.
  - Non-busy client statically bound to a fixed server

- Mode IV: Dynamic Self RR
  - Both busy and non-busy clients select servers by using a local RR.
LOAD BALANCING (Contd.)

- Mode V: Dynamic Central RR

- All client requests are sent to centralized dispatcher
- Dispatcher uses (global) RR to select server
Service time =10ms, Number of Clients=4, number of servers =2

<table>
<thead>
<tr>
<th>Strategy</th>
<th>System Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strgy 1</td>
<td>16.67</td>
</tr>
<tr>
<td>Strgy 2</td>
<td>25.64</td>
</tr>
<tr>
<td>Strgy 3</td>
<td>21.74</td>
</tr>
<tr>
<td>Strgy 4</td>
<td>21.51</td>
</tr>
<tr>
<td>Strgy 5</td>
<td>13.79</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strategy</th>
<th>System Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strgy 1</td>
<td>18.54</td>
</tr>
<tr>
<td>Strgy 2</td>
<td>27.50</td>
</tr>
<tr>
<td>Strgy 3</td>
<td>23.59</td>
</tr>
<tr>
<td>Strgy 4</td>
<td>23.35</td>
</tr>
<tr>
<td>Strgy 5</td>
<td>15.55</td>
</tr>
</tbody>
</table>

Improvement (%)

- 48.30% 27.21% 25.93% -16.15%
Service time = 80 ms, Number of Clients = 4, number of servers = 2

<table>
<thead>
<tr>
<th>Case</th>
<th>TP1 (Busy client)</th>
<th>TP2 (Non-Busy client)</th>
<th>TP (System)</th>
<th>Improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>5.04</td>
<td>1.49</td>
<td>6.53</td>
<td>-</td>
</tr>
<tr>
<td>II</td>
<td>5.39</td>
<td>1.36</td>
<td>6.75</td>
<td>3.3%</td>
</tr>
<tr>
<td>III</td>
<td>5.06</td>
<td>1.34</td>
<td>6.40</td>
<td>-2.1%</td>
</tr>
<tr>
<td>IV</td>
<td>5.02</td>
<td>1.31</td>
<td>6.32</td>
<td>-3.2%</td>
</tr>
<tr>
<td>V</td>
<td>5.00</td>
<td>1.36</td>
<td>6.36</td>
<td>-2.7%</td>
</tr>
</tbody>
</table>
LOAD BALANCING RESULTS

Effect of Service Time

Strategy Comparison (based on Strategy 1)

- 10ms
- 20ms
- 40ms
- 60ms
- 80ms
- 120ms

Strategic Comparison:
- 20.00%
- 10.00%
- 0.00%
- 10.00%
- 20.00%
- 30.00%
- 40.00%
- 50.00%
- 60.00%

Strategies: 1, 2, 3, 4, 5
- More complex System - 8 Clients and 4 Servers

![Effect of Service Time Graph]

**LOAD BALANCING-RESULTS**
- Static symmetrically balanced system gave the best performance
  - needs a priori knowledge of client service requirements
  - assumes client behavior is in one class (busy or non-busy)

- Half Round Robin Vs. Full Round Robin
  - Both strategies gave a comparable performance

- Performance differences among strategies more pronounced at lower server demands

- Central Vs. Local Round Robin
  - For smaller systems (sub-systems) local RR is preferable
    - central dispatcher incurs extra message overhead
  - For more complex system the centralized RR gives a superior performance