A Robust Persistent Storage Architecture for ACE

Sivaprasath Murugeshan
MS Thesis Defense
Dec 3, 2002

Committee:
Dr. Jerry James (Chair)
Dr. Arvin Agah
Dr. Susan Gauch
Overview

- Pervasive Computing
- ACE project
- Background
- Design
- Implementation
- Properties of the system
- Conclusions and Future work
- Related work
Pervasive Computing

- Diverse computing environment
- Myriad devices
- Storage and Computation distributed across heterogeneous network
- Robust and user-friendly
- Devices, storage, computation processes transparent
- Research challenges
  - Storage architecture, low-latency network protocols, etc
ACE Project

- Solution to Pervasive computing
- Smart rooms
- Personal workspaces
- Embedded devices
- ACE Services
- ASD – ACE Service Directory
Persistent Store

- User contexts should survive failures
- Objects – uninterpreted bytes
  - Text files, binary files, user contexts, etc
- Namespace – collection of objects
- Robust and highly available
- Consistent view
Overview

- Pervasive Computing
- ACE project
- **Background**
  - Design
  - Implementation
  - Properties of the system
  - Conclusions and Future work
  - Related work
Background

- Replication of services
- Well-defined interface to clients
- Failures in parts of the system
- Data consistency
- Synchronization among servers
- Servers being aware of status of other servers
- Organization of stored data
Consistency Model

- Data consistency in distributed systems
- Semantics of the abstraction provided by the store
- ‘read’ and ‘write’ operations
- Set of acceptable orderings
- Strong and weak consistency models
  - Correctness vs Performance
  - Examples
Strong consistency models

• **Strict Consistency**
  - Strongest consistency model
  - Global clock
  - Non-zero propagation delay. So, impractical

• **Linearizability**
  - Operations ordered in some sequential fashion consistent with read-write semantics
  - Non-overlapping operations ordered in the same way as real-time ordering

• **Sequential Consistency**
  - Restriction on non-overlapping operations removed

• **Desired programming model – deciding factor**
Failures

- **Machine failures**
  - Crash failures
  - Disk failures
  - Denial of service attacks

- **Network failures**
  - Message loss and corruption
  - Network partitions

- **Degree of robustness**
  - Types of failures that are detected
  - Recovery mechanisms
Programming Model

- Concurrent execution of tasks
- Multithreaded model
  - Different threads perform independent tasks
  - Easier to design
  - Difficult to debug
- Event-driven model
  - Server behaves like a finite state machine
  - Event handlers
  - Difficult to design
Issues in multithreading

- Mutual exclusion
  - Locks to protect shared data structures
  - Programmer’s responsibility

- Deadlock
  - Circular wait
  - Programmer’s responsibility

- Starvation
  - Same thread keeps acquiring the lock
  - Design of thread scheduler
Properties

- **Safety**
  - System does not do anything wrong
  - Deadlock freedom

- **Liveness**
  - System does something right
  - Starvation freedom

- **Behavior of the server**
Overview

- Pervasive Computing
- ACE project
- Background

**Design**
- Implementation
- Properties of the system
- Conclusions and Future work
- Related work
Design

- Peer-to-peer server architecture
- Objects and Namespaces in store
Client

- Services offered to client
  - Object commands
    - store_object
    - retrieve_object
    - store_unique_object
    - delete_object
    - list_objects
  - Namespace commands
    - create_namespace
    - delete_namespace
    - clear_namespace
    - list_namespaces
Server

- Client discovers server address from config. files
- Client randomly selects a server
- Concurrent processing of multiple client and server requests
Store

- Any non-volatile storage can be used
- Collection of objects and namespaces
Object commands

- store_object – store named object in the namespace
  - namespace
  - name
  - object
  - replication flag
- retrieve_object – retrieve named object from the namespace
  - namespace
  - name
- list_objects – list all objects in the namespace
  - namespace
Object commands

- **store_unique_object** – choose a unique name and store the object in the namespace
  - namespace
  - object
  - replication flag

- **delete_object** – delete named object from the namespace
  - namespace
  - name
Namespace commands

- `create_namespace` – create a namespace
  - namespace
- `clear_namespace` – delete all objects, but namespace remains
  - namespace
- `delete_namespace` – delete the namespace and all objects
  - namespace
- `list_namespaces` – list all namespaces
Consistency Model

• Linearizability
  • Example 1
    – P1: \( w(x)_1 \)
    – P2: \( r(x)_0 \quad r(x)_1 \)
  • Example 2
    – P1: \( w(x)_1 \)
    – P2: \( r(x)_0 \quad r(x)_1 \)

• Local property
  • Every object is linearizable => system is linearizable

• Two-phase commit protocol
Restart Mechanism

• Recovery after failure
• Incarnation File
  • stored in a specific location in the server machine
  • contains incarnation number
  • deleted during normal shutdown
• Incarnation Number
  • set to 0 when file is created
  • incremented after recovery
  • included with every message for updating store
  • checked before updating the store
Server joining and leaving

- **Joining**
  - has to be atomic
  - two-phase commit needed
  - client requests not processed during joining

- **Leaving**
  - crash detected by *sigpipe* handler
  - two-phase commit not necessary
Overview

- Pervasive Computing
- ACE project
- Background
- Design
- **Implementation**
  - Properties of the system
  - Conclusions and Future work
  - Related work
Two-phase commit

**Sender**

- **Ready**
  - Can operation be performed?

**Receiver**

- **Response**
  - Yes (locked object)
  - No (can’t lock object)

- **Decision**
  - Commit (Yes from all)
  - Abort (No from at least one)

- **Update (if Commit)**
  - Perform operation on object
Failure Detection

- Implementation using TCP/IP sockets
- Crash Failures
  - EPIPE error with socket related system calls
  - SIGPIPE handler invoked
- Disk Failures
  - Unable to perform disk I/O operations
  - Inform peer servers
- Status of peer servers updated
Data Structures

- Namespace Hash table
- peer_attributes
  - Peer identifier
  - Peer state
  - Socket id
  - Thread id
  - Incarnation number
- cond_var_array
  - Condition variable
  - Associated mutex variable
  - Flag1 (used or not)
  - Flag2 (status of two-phase commit)
- Object linked list
- client_request_list
  - Request type
  - Request parameters
  - Object
  - Incarnation number
  - Index in cond_var_array
Mutex and Condition variables

- Mutex variables
  - mutex_peer_attributes
  - mutex_client_request_list
  - mutex_cond_var_array
  - mutex_hash_table

- Condition variables
  - cond_peer_join
  - cond_var_array
Main thread

- initialize peer_attributes
- initialize array of condition variables
- initialize hash table
- install signal handlers
- update incarnation file
- create client_receive, peer_send and peer_receive threads
client receive

get requests from clients
parse the request
if server need not inform peers
    do local i/o and respond to the client
else
    add the request to client_request_list
wait for the result of two-phase commit
if signaled and two-phase commit is success
    do local i/o and inform the client of success
else
    inform the client of failure
peer send

read client_request_list
initiate two_phase_commit
wait for responses from peers (finite wait using select)
receive two_phase_commit_yes or two_phase_commit_no
send commit message or abort message
signal condition variable
delete request from client_request_list
peer receive

check peer_attributes
start two_phase_commit if not already initiated by another server
create child thread

update peer_attributes

forever
do
receive request from peer server
case request_type:

incarnation_number : update peer_attributes with incarnation number

i_am_dead : update peer_attributes
terminate this thread

peer_server_dead : update peer_attributes
terminate receive thread corresponding to the dead peer
peer receive

two_phase_commit_ready :
  - parse the request
  - acquire lock for namespace or object
  - break ties based on server id
  - send two_phase_commit_yes or two_phase_commit_no

two_phase_commit_commit :
  - receive the object
  - do local i/o
  - if disk failure, send i_am_dead message.
  - release lock for namespace or object

two_phase_commit_abort :
  - release lock for namespace or object.

done
Directory structure of the store

- Store - specific directory in the server machine
- Namespaces – subdirectories
- Objects - files
Overview

- Pervasive Computing
- ACE project
- Background
- Design
- Implementation

- Properties of the system
- Conclusions and Future work
- Related work
Assumptions

- **Thread package (Linux pthreads library)**
  - The thread scheduler is starvation free
  - Creating a child thread does not block
  - Terminating a child thread does not block

- **Communication mechanism (TCP/IP sockets)**
  - All messages that are sent are eventually delivered when there is no crash.
    Messages are not lost, corrupted or misdirected
  - Every crash is eventually detected
  - We have a perfect failure detector. So, all detected crashes are crashes
Invariants

- All shared data structures are protected by locks.
- Deadlock does not occur
  - No instance of circular wait in acquiring mutex variables
- Any thread that holds the lock does not block
  - No thread does infinite wait
Invariants

• The number of ‘peer_receive’ threads will eventually be the same as the number of servers set 'alive' in peer_attributes
  • peer_receive thread updates peer_attributes
  • peer_receive thread cancelled when peer server is set ‘dead’

• No server joins the group when a two-phase commit that has been initiated by a server for serving client request is in effect.
  • peer_send does two-phase commit in a sequential order
  • Processing either client requests or server joining requests
Properties

- Client requests are eventually served if the mutexes are starvation free and at least one server is alive and no server crashes
  - peer_send does timed wait using ‘select’ call
  - client_receive does timed wait on condition variable
- When there is a perfect failure detector and there are no network failures, the state of the persistent store including current state and pending commits, will be the same in all servers that are alive
  - Pending commit – namespaces and objects locked
  - State of the store changes only after successful two-phase commit
Properties

- Consistency is guaranteed by the two-phase commit protocol. Operations on the persistent store are linearizable
  - Linearizability is a local property
  - Writes are in some sequential order, same in all servers
  - Operations performed after acquiring locks
  - Ties in acquiring locks are resolved based on IP address
  - Commit is done in the same order in every server
  - Sequence of writes same order in every server
Limitations

- Network partitions
  - ‘alive’ servers considered to be ‘dead’
  - Results in inconsistencies

- Denial of service attacks
  - Servers flooded with requests from clients
  - Impairs performance of the server

- Two-phase commit protocol may block
  - Server crashes at inopportune moments
Conclusions and Future work

• Conclusions
  • Persistent storage architecture designed and implemented
  • Proved properties

• Future work
  • Different Network Protocols
  • Different Consistency Models
  • Security Issues
Related work

- Ninja
- Nile
- Websphere
- Weblogic
- Local consistency (Ahamad et al)
- Linearizable objects (M.P. Herlihy and J. M. Wing)
# Response time

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>List Namespaces</td>
<td>611 usec</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1542 usec</td>
<td></td>
</tr>
<tr>
<td>List Objects</td>
<td>1560 usec</td>
<td></td>
</tr>
<tr>
<td></td>
<td>218 usec</td>
<td></td>
</tr>
<tr>
<td>Retrieve Object</td>
<td>601 usec</td>
<td>(25 KB)</td>
</tr>
<tr>
<td>Create Namespace</td>
<td>46412 usec</td>
<td></td>
</tr>
<tr>
<td></td>
<td>17388 usec</td>
<td></td>
</tr>
<tr>
<td>Clear Namespace</td>
<td>74148 usec</td>
<td></td>
</tr>
<tr>
<td>Delete Namespace</td>
<td>21686 usec</td>
<td></td>
</tr>
<tr>
<td>Store Object</td>
<td>55771 usec</td>
<td>(25 KB)</td>
</tr>
<tr>
<td></td>
<td>68872 usec</td>
<td>(171 KB)</td>
</tr>
<tr>
<td></td>
<td>63578 usec</td>
<td>(25 KB)</td>
</tr>
</tbody>
</table>
Questions