Continuously Available Scalable Objects (CASO): An Elaboration of Fault-Tolerant CORBA

July 17th, 2002

Real-time and Embedded Distributed Object Computing Workshop
Arlington, VA USA
July 15-18, 2002

Russell L. Carter
FTOCO, LLC.
Prescott, AZ
(928) 541-0903
rcarter@ftoco.com
CASO Goals

■ Primary
  – Maximize throughput
  – Minimize latency
  – Minimize footprint
  – Support all servant state semantics

■ Secondary
  – Standards compliance (i.e., FT-CORBA)
Design Forces

- Stateful servers not supported by FT-CORBA Standard
  - Compliance does not support exactly once invocation semantics

- Fully consistent failure modes imply tightly coupled protocols
  - Some implementations do not permit load balancing of replicas, impairing performance of aggregated resources
  - Implementations require master/slaves relationship

- Need to support for protocols such as SSLIOP
**CAP Principle**

- Brewer and Fox conjecture, given:
  - **C**: Strong consistency
  - **A**: High Availability
    (Here interpreted as the ability of a client to complete a request)
  - **P**: Partition-resilience

- Pick at most two.

- Similar notions explored by Yu and Vahdat, at Duke.
**Multiple Invocations Problem**

- First observed by Frolund and Guerraoui in 1999, then Sebastian Wilhelmi of U. Karlsruhe in 2001

- Impossible for compliant implementation to correctly handle failure of stateful servant replica once state modified

- Implies only servants that can be made fault tolerant with FT-CORBA are those whose invocations are atomic
Example IDL Interface

// IDL

interface CheckingAccount {
    exception InsufficientFunds {};

    void debit (in long amount) {
        raises InsufficientFunds
    };
};
C++ Implementation

// C++

class CheckingAccount_i :: public POA_CheckingAccount {
public:
void debit (CORBA::Long amount)
throw (CORBA::SystemException, InsufficientFunds)
{
    if ((balance_-amount)<0)
        throw (InsufficientFunds ());
    balance_ -= amount;
    db_manager_.commit (balance_);
}
private:
    CORBA::Long balance_;
    DatabaseManager& db_manager_;
Invocation Operations Sequence

Client sees one commit, but servants commit twice.
Exactly Once Semantics

- FT-CORBA specifies reinvocations are transparent to the client (and server replicas).

- Servants need to know if an invocation is actually a reinvocation
  - If not a reinvocation, fine
  - if reinvocation, take action as appropriate, there are many possible policies

- FT-CORBA provides the required bits

- FT_REQUEST service context.
  - Accessible from a ServerRequest Interceptor
  - Implies that FT infrastructure must be linked with the code (for performance)
ReplicationManager: a Replicated Servant

- FT-CORBA specifies that the ReplicationManager (RM) is replicated

- Each RM maintains mutable state:
  - locations,
  - properties,
  - member object references
  - factories

- The RM itself is not immune to the problem of multiple invocations
Reinvocation Handling

- Basic idea: avoid unnecessary duplicate computations on reinvocation
  - Failed replica may have completed and communicated results to the replica group
  - (or it may not)

- Approach: view invocations on replicated servants as distributed database transactions
  - Each servant communicates the state of an active transaction via prepare () and commit ()
  - Each active transaction is associated with a unique Transaction Id (TID)

- Use FT_REQUEST service context for Transaction ID
Distributed Transactions and the CAP Principle

- Commercial databases are market driven to provide highest possible availability.
  - Accept a certain amount of inconsistency in face of failures
  - Provide high level tools to reinstate consistency after corruption

- Partition resilience provided by group protocols.
  - Reliable Broadcast, Consensus, Total Order Broadcast
  - Consensus uses majority vote to commit

- Design principle: Maximizing availability requires some inconsistencies must be allowed
- Tools must be provided to repair any that occur
Distributed Transactions and Group Protocols

- Group communication systems provide various orderings
  - FIFO
  - Causal
  - Total Order

- Databases use serializability
  - Implies total serial order,
  - In commercial best practice, non-conflicting transactions processed concurrently

- CAP Principle:
  - Simplistically layering database semantics over total order group communication enforces consistency at the expense of availability

- Need to integrate group communication with transaction semantics
Practical Distributed Transaction Implementations

- Two recent efforts:
  - Kemme and Alonso
  - Amir and Tutu

- Eager Database Replication
  - Based on Total Order and Reliable Broadcast subsystems
  - Explicitly support relaxed consistency algorithms

- Relaxed consistency algorithms well understood in database community
  - Examples: Replication with Cursor Stability, and Snapshot Isolation
  - Read/write conflicts significantly reduced or eliminated
  - Possible inconsistencies, but tightly characterized, localized to failed replica
    - Inconsistencies are repaired upon replica recovery

- This approach enables concurrent processing, maximizing availability
Replication with Serializability

The lock manager of each node N controls and coordinates the operation requests of transaction $T_i$:

1. **Local Reading Phase**: Acquire local read lock for each read operation $r_i(X)$. Defer write requests $w_i(X)$ until end of reading phase.

2. **Send Phase**: If $T_i$ is read-only, then commit, else bundle all writes in $WS_i$ and TOBcast to group.

3. **Lock Phase**: Upon delivery of $WS_i$, request all locks for $WS_i$ in an atomic step:
   
   (a) For each operation $w_i(X)$ in $WS_i$:

   (i) If there is no lock on $X$, grant the lock.

   (ii) If there is a write lock on $X$ or all read locks on $X$ are from transactions that have already processed their lock phase, enqueue the lock request.

   (iii) If there is a granted read lock $r_j(X)$ and the write set of $WS_j$ of $T_j$ has not yet been delivered, abort $T_j$ and grant $w_i(X)$. If has already been sent, RBcast abort message $a_j$.

(b) If $T_i$ is a local transaction, rbcast commit message $c_i$. 
Replication with Serializability (cont.)

(4) **Write Phase:** Whenever a write lock is granted perform the corresponding operation.

(5) **Termination Phase:**

(a) *Upon delivery of a commit message* $c_i$: Whenever all operations have been executed commit $T_i$ and release all locks.

(b) *Upon delivery of an abort message* $a_i$: Undo all operations already executed and release all locks.

Extracted from: Kemme and Alonso: *A New Approach to Developing and Implementing Eager Database Replication Protocols*
Algorithm Characteristics

- Serializability
  - deadlock free, 1-copy-serializable, guarantees transaction atomicity
  - Problem: aborts readers

- Replication with Cursor Stability
  - release read locks immediately after reading (rather than after commit)
  - Inconsistencies: non-repeatable reads, read skew, and write skew may occur in failed replicas

- Snapshot Isolation
  - Uses sequence numbers for BOT (begin of trans) and EOT timestamps to uniquely determine snapshots
  - write skew may occur in failed replicas
Integration with FT-CORBA

- Uses FT_REQUEST service context to generate Transaction ID
- Upon invocation, a ServerRequest Interceptor grabs the FT_REQUEST and places in TSS for later use by the CASO_Manager
- Each Servant implements a Transactional Resource Object for each IDL defined method.
  - inherits from ReadOnlyResource for read only methods
  - inherits from WriteResource for mutable methods
  - Each Resource implements a start() method to initialize the Resource with method parameters
  - Each Resource implements a finish() method for return, inout, out values, or exceptions
ReadOnly Resource

- Programmer implements the virtual method prepare()
  - Invoked by the TransactionManager
  - Sets the return value(s) or stores an exception
WriteResource

- Programmer implements two pure virtual methods invoked by the TransactionManager:
  - `prepare()`: uses the parameters supplied by `start()` to assemble the writeset and store result(s) or an exception
  - `commit()`: invokes the configured replication algorithm on the writeset

- **Writeset marshalling**
  - Helper methods provided by template:
  - `class SetDefaultProperties_tr : public WriteResource<Properties> {
    public:
    
    typedef Properties WritesetType;
    typedef WriteResource<WritesetType> Inherited;
    
    - **Usage:**
      
      Inherited::set_writeset (props);

    - Also `get_writeset (WriteSetType& ws) helper method.`
Transaction LockManager

- Application requests a Lock with LockManager::get_lock().
- Locks are assembled into a LockSet (std::list) and provided to each Resource during Resource initialization.
- Locks are of arbitrary scope
  - Single Lock shared among all Resources
  - Multiple Locks shared arbitrarily
- Locking can be as fine or coarse grained as required.
- CASO_Manager implicitly manages all lock ops associated with a given Resource.
- Application sole responsibility is determining initial distribution of locks among resources.
Resource Usage

- CASO_Manager performs distributed write transaction
- First retrieves TID from TSS and associates it with this transaction
- If transaction is reinvocation
  - If already committed, just returns committed writeset
  - If transaction in progress, aborts current instance and restarts with new writeset
- CASO_Manager acquires and releases Read and Write Locks for Resource
- Sets new writeset and returns

```cpp
this->set_default_properties_tr_.start (props);
this->caso_mgr_.execute_write_transaction
    (this->set_default_properties_tr_);
this->set_default_properties_tr_.finish ();
```
CASO Organization

Computer System A

- Server Process
  - ReplicationManager
- CASO_Manager
- CASO_Manager
  - App Object
  - App Object

Computer System B

- Server Process
  - App Object
- CASO_Manager
- ReplicationManager
- CASO_Manager
  - App Object
  - App Object
Group Communication Instance

Consensus → RSend

TOBcast → Consensus

ORB

RBcast

SMessage op

IDL op
Conclusions

- Each location processes invocations and distributes the results to the group (tobcast+rbcast)
- There is no notion of “primary”, each location commits non-conflicting requests in received order
- Conflicts may require abort/retry
  - Well understood database consistency strategies available
  - Increase availability by decreasing conflicts, may require replica repair
- Lock scope is entirely up to the application programmer
- Easily supports per object/method locking
- Non-conflicting requests are processed concurrently
- Load balancing FT requests now very easy
- Multiple invocation problem eliminated
- No interference with protocols such as SSLIOP
Acknowledgements

The author greatly appreciates the support of Navy SBIR Phase II contract N00178-97-C-3023, and in particular Dr. Bruce R. Copeland (CopelandBR@nswc.navy.mil).