REPLICATED VS. CENTRALIZED MODEL SHARING

Prasun Dewan
Department of Computer Science
University of North Carolina at Chapel Hill
dewan@cs.unc.edu
Code available at: https://github.com/pdewan/ColabTeaching
How to implement model-based sharing?
How do build higher level abstractions for model-based sharing?
More than one architecture to implement this user interface
Single-user Architecture

Interactor

Model
What are these architectures, what are their pros and cons?
REPLICATED

Interactor

Model

Interactor

Model

Interactor

Model
# Single-User vs Replicated Algorithm: Running Example

## UI Thread

- For each input `I`
  - `I` should be followed by matching `ListEditInput`, `ListEditMade`, `ListEditNotified`, `ListEditObserved`, `ListEditDisplayed`
  - For each replica, `I` should be followed by matching `ListEditSent to Others`

## Receiving Thread

- For each `ListEditReceived R`
  - `R` should be followed by matching `ListEditMade`, `ListEditNotified`, `ListEditObserved`, `ListEditDisplayed`
  - For each replica, `R` should be followed by matching `ListEditSent`
General Model-Interactor Pattern: from ListEdit ops to Edit Ops

For each input I
I should be followed by matching EditInput, EditMade, EditNotified, EditObserved, EditDisplayed
For each replica, I should be followed by matching EditSent to Others

For each EditReceived R
R should be followed by matching EditMade, EditNotified, EditObserved, EditDisplayed
For each replica, R should be followed by matching EditSent
**Single-user vs Replicated (Cycles)**

- **History**
  - **elementAdded**
  - **observableAdd**
  - **input**
    - **EchoerInteractor**

- **Replicated History**
  - **toOthers**
    - **Communicator**
      - **observableAdd**
      - **objectReceived**

- **IMInteractor**
  - **input**
  - **observableAdd**

- **IMCoupler**
**Single-user vs Replicated: Replicated and Non Replicated Write Methods**

Each write method has a single-user and replicated version.

Both notify observers.

Local input triggers replicated version.

Remote input triggers single-user version.

Multiple write methods `setP()`, `replicatedSetP()`.

Element added `elementAdded` to `observableAdd` in `EchoerInteractor`.

Replicated `History` triggers `toOthers` in `Communicator`.

Local input triggers replicated `History`.

Remote input triggers single-user version.

Input triggers `replicatedAdd` in `IMInteractor`.

Multiple `observableAdd` for `replicatedAdd` in `IMCoupler`.

History data is replicated and non-replicated.
**Single-user vs Replicated: Can Detect if Incoming Event has Been Processed before**

- **History**
  - `elementAdded`
  - `observableAdd` to `EchoerInteractor`

**EchoerInteractor**

- Notification has information about old state of changed object
- Can determine if this object is at old state or new state
- Write method not called if object is at new state

**Replicated History**

- `observableAdd`

**IMInteractor**

- `toOthers`

**Communicator**

- `observableAdd` to `Replicated History`
- `objectReceived` to `IMCoupler`

**IMCoupler**

- `objectReceived`
**MODELS VS. NOTIFICATIONS**

- **Beans**
  - Property collections
  - Differ in properties

- **Lists**
  - Variable length indexed lists
  - Differ based on subsets of list operations

- **Table model** is another important kind not needed in this course

- **Actions**
  - **Received message contains new property value**
    - Assuming no side effects
    - If new property value same as current property value, do not call write method
  - **Received message can contain new size**
    - If object is of new size, do not call write method
  - **Received message contains new key value**
    - If new key value same as old, do not call write method
SINGLE-USER vs. MULTI-USER STEPS
GENERAL MODEL-INTERACTOR PATTERN: FROM LISTEDIT ops to EDIT Ops

UI Thread

For each input I

I should be followed by matching EditInput, EditMade, EditNotified, EditObserved, EditDisplayed

For each replica, I should be followed by matching EditSent to Others

Receiving Thread

For each EditReceived R

R should be followed by matching EditMade, EditNotified, EditObserved, EditDisplayed
Consistency issues of causality and concurrent operations (to be addressed later)
Multiple physical models represent a single logical model
Replicated Model-Interactor Algorithm

UI Thread

For each input I
I should be followed by matching EditInput, EditMade, EditNotified, EditObserved, EditDisplayed

For each replica, I should be followed by matching EditSent

Receiving Thread

For each EditReceived R
R should be followed by matching EditMade, EditNotified, EditObserved, EditDisplayed

Problems?
Replication Guarantee

UI Thread

For each input I

I should be followed by matching EditInput, EditMade, EditNotified, EditObserved, EditDisplayed

For each replica, I should be followed by matching EditSent

Receiving Thread

For each EditReceived R

R should be followed by matching EditMade, EditNotified, EditObserved, EditDisplayed

Each model executes the same set of operations

Not the same sequence!

Consistency issues of causality and concurrent operations (to be addressed later)
Assume Stronger Guarantee

For each input I
I should be followed by matching EditInput, EditMade, EditNotified, EditObserved, EditDisplayed

For each replica, I should be followed by matching EditSent

For each EditReceived R
R should be followed by matching EditMade, EditNotified, EditObserved, EditDisplayed

Assume that each model executes the same sequence of operations

Performance issues?
Correctness issues?
Performance

Input n

Interactor

Model

Find prime factors

Interactor

Model

Find prime factors

Inefficient replicated computations!

Computation vs. communication costs
**Reading a Centralized External Resource**

![Diagram]

- **Input load**
  - **Interactor**
  - **Model**
    - `read f`
  - **File f**

**Bottleneck!**
PROBLEMS

UI Thread

For each input I

I should be followed by matching EditInput, EditMade, EditNotified, EditObserved, EditDisplayed

For each replica, I should be followed by matching EditSent

Receiving Thread

For each EditReceived R

R should be followed by matching EditMade, EditNotified, EditObserved, EditDisplayed

Assume that each model executes the same sequence of operations

Multiple computations and bottlenecks

Correctness issues?
Is executing the same operation multiple times equivalent to executing the operation a single time?
Reading a Centralized External Resource

Model

Interactor

Model

Interactor

read f

read f

File f

a

Change read to write?
Writing to a Centralized External Resource

Each replica writes to the file!

Behavior of centralized and replicated different

Write is not idempotent

Executing idempotent operations once is the same as executing them multiple times, operation is a function of only its arguments

Assumption: Only idempotent operations

Input b

Interactor

Model

Write f, b

File f

abb

Interactor

Model

Write f, b
Other examples of idempotent operations in practice?
**Sending Mail Together**

Do not always have the option of replicating resources.

Assumption: Only idempotent operations.
Replicated Model: Issues

Consistency issues of causality and concurrent operations (to be addressed later)

Correctness and performance issues when model is non deterministic, accesses central resources, and has side effects
ReplIcated VS CenTralized

- Interactor
  - Model
  - Interactor

- Interactor
  - Model
  - Interactor

- Interactor
  - Model
  - Interactor

Arrows indicate the flow of information or processes between the components.
CENTRALIZED SESSION MEMBER TYPES

- Interactor
- Model
- Interactor
**Single-User vs Centralized Algorithm: Running Example**

- **Slave UI Thread**
  - For each input \( I \)
  - \( I \) should be followed by matching ListEditInput and ListEditSent to Master

- **Master Receiving Thread**
  - For each ListEditReceived \( R \)
  - \( R \) should be followed by matching ListEditMade, ListEditSent to Others

- **Slave Receiving Thread**
  - For each ListEditReceived \( R \)
  - \( R \) should be followed by matching ListEditDisplayed
None of the replication issues
Feedback times involve round trip delays
Feed through incurs extra hop (beyond relaying)
Refresh and query operations also involve round trip delays (e.g. searching history)
Can we fix the last problem?
Caching!
CACHING VS. REPLICA

Model cache is data repository without side effects

Updated in response to distributed messages from real central model

No divergence of caches, real model has the real state

Like real model it fires events to local observers

Write operations require round trip

Read operations access local data
Each interactor is distribution and collaboration aware: it sends messages to central model.

As is model cache, it receives messages from central model.
DISTRIBUTION UNAWARE INTERACTOR WITH MODEL CACHE/PROXY

Model cache is a proxy that forwards interactor operation without changing its data.

Less distribution awareness and more automation.

Model cache is still distribution aware, both sending and receiving messages.

Some distribution awareness is necessary in application if we use general purpose group communication layer.
EXAMPLE CENTRALIZED ALGORITHM (NO CACHING)

Slave UI Thread

For each input I

I should be followed by matching ListEditInput and ListEditSent to Master

Master Receiving Thread

For each ListEditReceived R

R should be followed by matching ListEditMade, ListEditSent to Others

Slave Receiving Thread

For each ListEditReceived R

R should be followed by matching ListEditDisplayed
**Example Centralized Algorithm (Caching)**

### Slave UI Thread

For each input I

I should be followed by matching ListEditInput, ListEditForwarded to Slave Proxy and ListEditSent to Master via Slave

### Master Receiving Thread

For each ListEditReceived R

R should be followed by matching ListEditMade, ListEditSent to Others

### Slave Receiving Thread

For each ListEditReceived R

R should be followed by matching ListEditMade, ListEditNotified in Slave and ListEditDisplayed
GENERAL CENTRALIZED ALGORITHM: LISTEDIT ➔ EDIT

For each input I
I should be followed by matching EditInput, EditForwarded and EditSent to Master via Slave

For each EditReceived R
R should be followed by matching EditMade, EditSent to Others

For each EditReceived R
R should be followed by matching EditMade, EditNotified in Slave and EditDisplayed in Slave UI Thread
Centralized Architecture
CACHING WITH GROUPMESSAGES

SlaveInteractor

Histoy

EchoerInteractor

SlaveHistory

Communicator

MasterHistory

Communicator

MasterCoupler

SlaveCoupler

Source is master rather than peer

Master puts name of original sender in marshalled edit operation used my slave coupler to compose new element value
REPPLICATED ADD: SIMPLE MARSHALLING

```java
public synchronized void replicatedAdd(ElementType anElement) {
    int anIndex = size();
    super.observableAdd(anIndex, anElement);
    if (communicator == null) return;
    ListEdit listEdit = new AListEditelementType(OperationName.ADD, anIndex, anElement, ApplicationTags.IM);
    communicator.toOthers(listEdit);
}
```

```java
public interface ListEdit<ElementType> extends Serializable {
    int getIndex();
    void setIndex(int anIndex);
    ElementType getElement();
    void setElement(ElementType anElement);
    ...
}
```
**Centralized Add: Aware Marshalling**

```java
public synchronized void centralizedAdd(ElementType anInput, String aClientName) {
    int anIndex = size();
    super.add(anIndex, anInput);
    UserEdit<ElementType> userEdit = new AUserEdit<ElementType>(OperationName.ADD, anIndex, anInput, ApplicationTags.IM, aClientName);
    communicator.toOthers(userEdit);
}
```

```java
public interface UserEdit<ElementType> extends ListEdit<ElementType>{
    public String getUserName();
    public void setUserName(String userName);
}
```
Motivating Unicast in Multicast Layer

- **whisper** `playername = your whisper here`
  
  ... so only the player(s) named, and in the room, can hear your whisper.

```java
public class ASlaveSimpleList<ElementType>
    extends ASimpleList<ElementType>
    implements SlaveSimpleList<ElementType> {
...

public synchronized void proxyAdd(ElementType anElement) {
    int anIndex = size();
    ListEdit listEdit = new AListEdit<ElementType>(
        OperationName.ADD, anIndex, anElement,
        ApplicationTags.IM);
    communicator.toClient(MasterIMModelLauncher.CLIENT_NAME, 
                      listEdit);
}
```
Replicated Model

Consistency issues of causality and concurrent operations (to be addressed later)

Correctness and performance issues when model accesses central resources, is non deterministic, and has side effects
**Centralized Model**

- All updates routed through central model and local cache’s guaranteed to be consistent
- Each operation executed only once, so no problems of non idempotent operations, central bottlenecks, and expensive operations
- Round trip delay to get local feedback
- Central bottleneck which may not always be available

Combine the advantages?  
Causal multicast and OT for consistency  
Multi operation execution?
A distinguished model executes operations that are expensive, non-idempotent, or access central resources.
A central model executes operations that are expensive, non-idempotent, or access central resources.

Both solutions are application-specific.
Can we automate these architectures?
**General Replicated Algorithm**

**UI Thread**

For each input $I$

$I$ should be followed by matching $\text{EditInput}$, $\text{EditMade}$, $\text{EditNotified}$, $\text{EditObserved}$, $\text{EditDisplayed}$

For each replica, $I$ should be followed by matching $\text{EditSent}$ to Others

**Receiving Thread**

For each $\text{EditReceived}$ $R$

$R$ should be followed by matching $\text{EditMade}$, $\text{EditNotified}$, $\text{EditObserved}$, $\text{EditDisplayed}$

For each replica, $R$ should be followed by matching $\text{EditSent}$
**Automatic General Centralized Algorithm**

**Slave UI Thread**

For each input I

I should be followed by matching EditInput, EditForwarded, and EditSent to Master via Slave

**Master Receiving Thread**

For each EditReceived R

R should be followed by matching to Others

**Slave Receiving Thread**

For each EditReceived R

R should be followed by matching EditMade, EditNotified in Slave and EditDisplayed
CENTRALIZED AND REPLICATED MODEL SHARING

Model Sharing

Group Communication (Multicast)

Interprocess Communication (Sockets, RMI, ..)

Goal is to reduce/eliminate the collaboration awareness in application code

Can assume language/compiler support
CENTRALIZED AND REPLICA TED MODEL SHARING (REVIEW)

Model Sharing

Group Communication (Multicast)

Interprocess Communication (Sockets, RMI, ..)

Goal is to reduce/eliminate the collaboration awareness in application code

Can assume language/compiler support
Original Example

Please enter an input line or quit or history
The woods are lovely dark and deep
[Alice] The woods are lovely dark and deep
Please enter an input line or quit or history
[Bob] But I have promises to keep
[Cathy] And miles to go before I sleep
history
[Alice] The woods are lovely dark and deep, [Bob] But I have promises to keep, [Cathy] And miles to go before I sleep
Please enter an input line or quit or history
Another Example

```java
public class AConcertExpense implements ConcertExpense {
    float unitCost = 0;
    int numberOfAttendees = 0;
    public float getTicketPrice() { return unitCost; }
    public void setTicketPrice(float newVal) {
        unitCost = newVal;
    }
    public int getNumberOfAttendees() { return numberOfAttendees; }
    public void setNumberOfAttendees(int newVal) {
        numberOfAttendees = newVal;
    }
    public float getTotal() {
        return unitCost*numberOfAttendees;
    }
}
```
AUTOMATING GENERAL REPLICATED ALGORITHM

For each input I
I should be followed by matching EditInput, EditMade, EditNotified, EditObserved, EditDisplayed

For each replica, I should be followed by matching EditSent to Others

For each replica, R should be followed by matching EditSent

Operations to be automated

Receiving Thread

For each EditReceived R
R should be followed by matching EditMade, EditNotified, EditObserved, EditDisplayed

For each replica, R should be followed by matching EditSent

Interception of operations and proxy generation possible with language/compiler support

Somehow need to distinguish between edit and non-edit model methods

Operations to be automated
Automatic General Centralized Algorithm

For each input I
I should be followed by matching EditInput, EditForwarded and EditSent to Master via Slave

Operations to be automated

For each EditReceived R
R should be followed by matching to Others

Proxy generation possible with language/compiler support

Somehow need to distinguish between edit and non edit model methods

For each EditReceived R
R should be followed by matching EditMade, EditNotified in Slave and EditDisplayed
**Write vs Non Write Methods**

- **Replica**
  - `float getTicketPrice()`
  - `void setTicketPrice(float)`

- **Collaboration-aware Proxies**
  - Need a way to select methods that are broadcast.
Programmer-Specified Broadcast Methods

float getTicketPrice ()

Replica (Model or Interactor)

void broadcast setTicketPrice (float)

Collaboration-aware Proxies

Cycles within the same method stopped by Colab

**Potential for Spurious Broadcasts**

void broadcast notify ()

Replica

void broadcast setTicketPrice (float)

Collaboration-aware Language Runtime

Must carefully choose which methods are broadcast

void broadcast notify ()

Collaboration-aware Language Runtime

void broadcast setTicketPrice (float)

Replica

void broadcast notify ()

Collaboration-aware Language Runtime

void broadcast setTicketPrice (float)

Replica
Can broadcast methods in any object, not just model.

In Xerox Colab applications, interactor methods were broadcast.
Multi-Layer Broadcast Increase Spurious Broadcast Problem

void broadcast.textChanged(evt)

void broadcast.setTicketPrice(float)

Interactor

Model

Must consider all calls to a method before making it broadcast.

Another solution to select methods to broadcast?
Somehow need to distinguish between edit and non edit model methods

In general an object is a blackbox and we do not know its write methods without explicit programmer specification
Restrict Model Types

- Lists
  - Variable length indexed lists
  - Differ based on subsets of list operations exposed

- Beans
  - Property collections
  - Differ in properties

- Table model
  - Key, Value Collections

**Bean Pattern/Conventions**

Typed, Named Unit of Exported Object State

```java
public class C {
    public T getP() {
        ...
    }
    public void setP(T newValue) {
        ...
    }
}
```

- **Bean**
  - Name P
  - Type T
  - Read-only
  - Editable
  - Getter method
  - Setter method

*Bean convention: For humans and tools*
Restrict Model Patterns

- Lists
  - Variable length indexed lists
  - Differ based on subsets of list operations exposed

- Beans
  - Property collections
  - Differ in properties

- Table model
  - Key, Value Collections

Can assume certain programming conventions for l such as for getters and setters to extract write methods

Reflection, introspection and proxy generation can then be used to broadcast/forward write methods and generate proxies and replicas

Works for a theoretically restricted set of model types

ISSUES

- How to determine methods to be broadcast?
- How to find corresponding replicas?
How are corresponding replicas found?

Assume model classes are singletons.

Assume replica programs instantiate the same sequence of objects.

Replicated shared window systems assumes same sequence of windows.

Can use names of Java windows

Register and lookup of names
CONNECTING REPLICA

- Order of object instantiation
- Names of objects
  - Works for AWT windows every window has a name
  - If two windows have the same name, assume they are the same
- Explicit remote lookup and register
  - Central registry used to connect objects with names
  - A la session manager and RMI registry
CENTRALIZED: CONNECTING PROXIES
Replicated: Connecting Replicas

Registry replica can be kept consistent and execute operations that need to be centralized (send mail, access file)

If already registered, no-op to allow symmetric programs
public class CounterServer {
    public static void main (String[] args) {
        try {
            Registry rmiRegistry = LocateRegistry.getRegistry();
            ConcertExpense concertExpense = new AConcertExpense();
            UnicastRemoteObject.exportObject(concertExpense, 0);
            rmiRegistry.rebind(ConcertExpense.class.getName(), counter);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
RMI LookUp

```java
public class CounterClient {
    public static void main (String[] args) {
        try {
            Registry rmiRegistry = LocateRegistry.getRegistry();
            ConcertExpense concertExpense = (ConcertExpense)
                rmiRegistry.lookup(ConcertExpensse.class.getName());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

RMI does not support centralized model sharing as it creates a pure proxy and not cache of model object and does not distinguish between read and write methods
package budget;
import bus.uigen.ObjectEditor;
import edu.unc.sync.Sync;
public class SyncBudgetSymmetric {
    static String SERVER_NAME = "localhost/A";
    static String MODEL_NAME = "demoBudget";
    public static void main(String[] args) {
        String[] syncArgs = {
            "--oe"};
        Object model = Sync.replicateOrLookup(
            SERVER_NAME,
            MODEL_NAME,
            AConcertExpense.class,
            args[0],
            syncArgs);
        ObjectEditor.edit(model);
    }
}
Server UI After Replicate
ALICE UI AFTER REPLICATE
Bob UI After Bob's Lookup
Alice UI After Bob’s Lookup
ISSUES

- How to determine methods to be broadcast?
- How to find corresponding replicas?
- When should write methods be called on corresponding replicas?
**When to Synchronize**

When both the sending and receiving application say synchronize

Sending site can say real-time synchronize to execute synchronize operation when a write method is executed at the sending site

Receiving site can say real-time synchronize to execute synchronize operation when a write method is received

Integrates synchronous and asynchronous (Dropbox, GoogleDrive, OneDrive) sharing
ONE SITE DISCONNECTED: IT SYNCs
RECEIVER IMMEDIATELY UPDATES

[Image of a computer interface showing a window with fields for Number Of Attendees, Ticket Price, and Total, with values entered and a checkbox for Real Time Synchronize checked.]
Both Sites Disconnected and Change
TOP USER SYNCs

[ConcertExpense]
File Edit View Customize ConcertExpense

Number Of Attendees
0

Ticket Price
23.5

Total
0.0

[ServerProxy]
File Edit View Customize ServerProxy

Synchronize
Real Time Synchronize

[ConcertExpense]
File Edit View Customize ConcertExpense

Number Of Attendees
3

Ticket Price
0.0

Total
0.0

[ServerProxy]
File Edit View Customize ServerProxy

Synchronize
Real Time Synchronize
NO UPDATE IN BOTTOM USER
**Bottom User Synchs**

![Software Interface](image1)

- **Number Of Attendees**: 0
- **Ticket Price**: 23.5
- **Total**: 0.0

![Software Interface](image2)

- **Number Of Attendees**: 3
- **Ticket Price**: 0.0
- **Total**: 0.0

![Software Interface](image3)

- **Synchronize**: Real Time Synchronize
BOTTOM USER HAS BOTH CHANGES
TOP USER SYNCS

[ConcertExpense]
File Edit View Customize
ConcertExpense

Number Of Attendees
0

Ticket Price
23.5

Total
0.0

[ServerProxy]
File Edit View Customize ServerProxy

Synchronize

Real Time Synchronize

[ConcertExpense]
File Edit View Customize ConcertExpense

Number Of Attendees
3

Ticket Price
23.5

Total
70.5
Both Users in Sync
SUMMARY

- Two ways to share a logical model among multiple users
  - Both involve local interactor at each user site
  - In replicated, symmetric models at each site
    - They service read and write methods for local interactors
    - Send updates to other models without necessarily waiting for them to be made.
    - Side effects can be executed multiple times and concurrent operations can lead to inconsistent
  - In centralized a centralized model at special (possibly user) site
    - Each site can cache the model for reads.
    - Writes wait until central model updates and are then cached at local site.
    - Cache stores only data and has no side effects
    - Indirection may require user-aware marshalling
SUMMARY

- Can automate model sharing

- Identifying methods to be broadcast
  - Broadcast keyword or annotation
    - Cascaded broadcasts
  - Fixed types
    - No programmer-defined types such as beans
  - Using conventions or patterns for describing models
    - Restricted models
  - Tradeoff: Cycles vs. restricted models

- Connecting corresponding objects
  - Sequence of objects instantiated
  - Singleton objects
  - Register/lookup

- When to broadcast
  - In general on Explicit/Implicit Sync
ABSTRACTIONS IN ACTUAL IMPLEMENTATION

- **Xerox Colab**
  - Broadcast methods, immediate execution, implicit replica binding

- **Sync**
  - Patterns and predefined shared string, record, sequence, explicit/implicit sync, register/lookup

- **LiveMeeting**
  - Predefined shared int, string, real

- **Google Hangout**
  - Single shared table

---


NEXT

Model Sharing

Group Communication (Multicast)

Interprocess Communication (Sockets, RMI, ..)

Non model sharing?