Semantic Aspects of Asynchronous RMI – the RMIX Approach

JPDC-2004. Santa Fe, New Mexico, Apr 26, 2004

Dawid Kurzyniec
Vaidy Sunderam

Dept. Of Math and Computer Science
Emory University, Atlanta, GA
{dawidk,vss}@mathcs.emory.edu

EMORY
UNIVERSITY

Remote Method Invocations

- One of the most popular remote communication paradigms
  - Hides communication complexity behind well-understood semantics of a method call
  - Simplifies development
  - Simplifies distributing legacy codes
- Performance impact
  - Caller blocked until invocation completes
  - Even when execution time is negligible (e.g. event notification), performance limited by network latency

Coping with RMI Inefficiencies

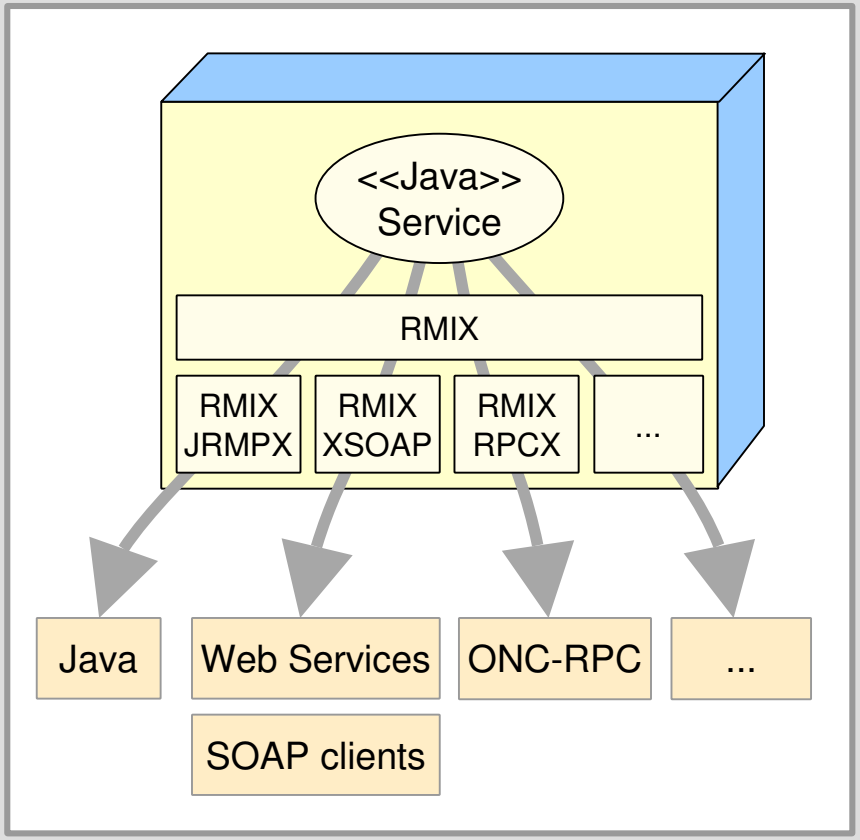
- Solution #1: use multiple threads of execution
  - Introduces concurrency explicitly
  - However, requires significant changes to the code
- Solution #2: non-blocking (asynchronous) RMI
  - Concurrency introduced implicitly
  - Raises subtle syntactic, semantic, and security issues...
    - Completion notification, parameter consistency, execution order, exception handling, cancellation, thread security contexts
  - ... Which are often neglected or not well understood
    - Most of existing approaches focus solely on performance

Outline

- This project: analysis of functional aspects of async RMI
  - Resulting specifications implemented and tested within the multiprotocol RMI framework termed RMIX
- Overview of RMIX
- Analysis
  - Invocation syntax
  - Semantics
    - Data consistency, execution order, exception handling, cancellation
  - Security
- Implementation status
- Conclusions

RMIX Overview

- Extensible RMI framework
- Client and provider APIs
  - uniform access to communication capabilities
  - supplied by pluggable provider implementations
- Multiple protocols supported
  - JRMPX, ONC-RPC, SOAP
- Configurable and flexible
  - Protocol switching
  - Invocation interceptors
  - ...



Async-RMIX Design Principles

- Enable but do not mandate
  - For providers: optional to implement
  - For clients: extension which retains full compatibility
- Precise semantics
  - If implemented, must adhere to certain constraints
- Server transparency
  - No changes to the server-side application code
- Retain RMI simplicity
- Principle of least surprise
  - if multiple options are possible, choose safe defaults
- Do not impede performance
- Avoid security vulnerabilities

Async RMI Syntax

- Problem: call result (return value or exception) not available immediately upon return of control
  - Must arrange for completion notification
  - Approaches: futures, callbacks, result queues
  - Some implementations avoid the issue by limiting support to one-way calls
- Solution: supplementary methods in remote stubs
  - Return futures; take callbacks as extra arguments
  - Introduced via “asynchronous interfaces”

Async RMI Syntax (example)

```
interface Hello extends Remote {
  void hello(String greeting) throws RemoteException;
}
```

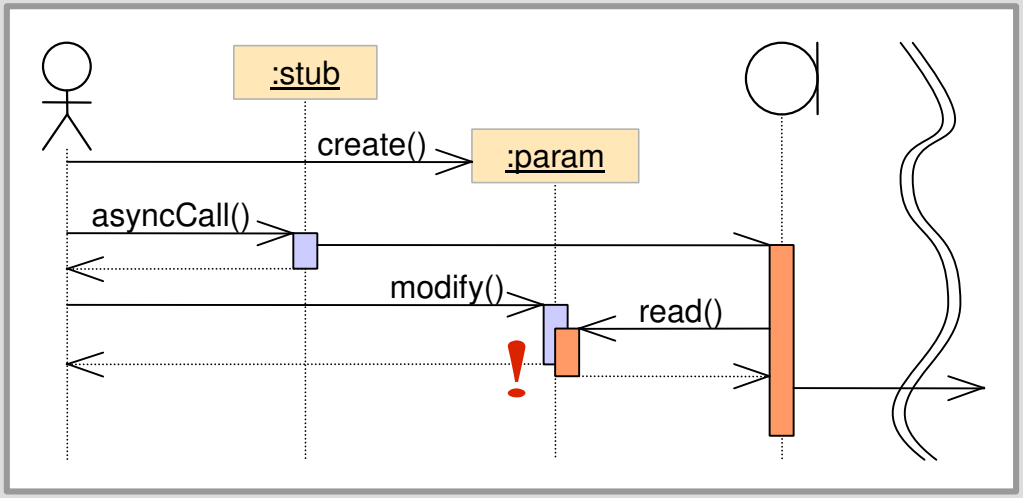
```
interface AsyncHello extends Hello {
  Future asyncHello  (String greeting) throws RemoteException;
  Future cbasyncHello(String greeting, Callback cb) throws RemoteException;
  void   onewayHello (String greeting) throws RemoteException;
}
```

```
AsyncHello hello = (AsyncHello)Naming.lookup(...);
Future f = hello.asyncHello(”World!”);
f.get();
```

```
interface Future {
  boolean isDone();
  Object get() throws InterruptedException, ExecutionException;
  Object get(long timeout, TimeUnit granularity)
    throws InterruptedException, ExecutionException, TimeoutException;
}

interface Callback {
  void completed(Object result);
  void failed(Throwable cause);
}
```

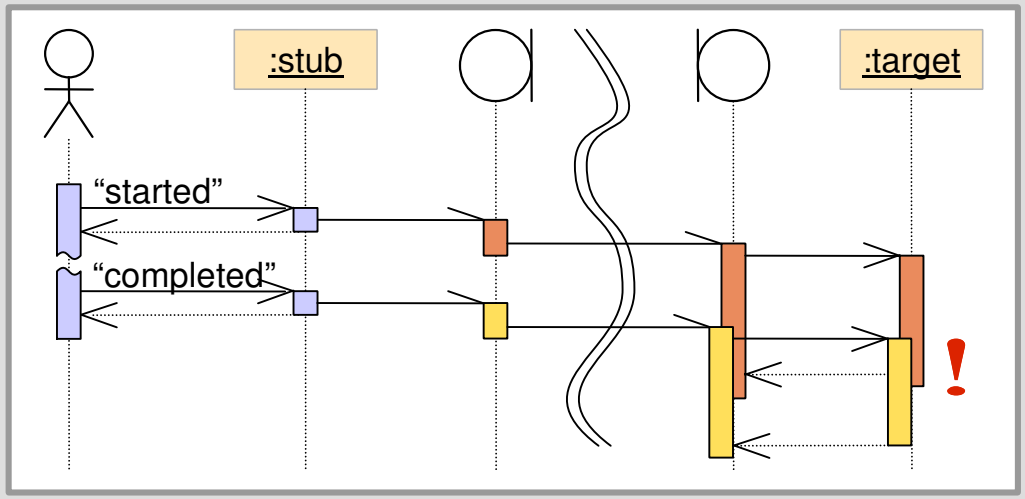
Parameter consistency



- Stages of remote call
  - call marshalling
  - awaiting response
  - result unmarshalling
- When should async call return control?
  - If after marshalling, may become blocking
  - If before fully marshalled, parameters are vulnerable to corruption (unless immutable)
  - (A.k.a. *synchronization mode,* e.g. in CORBA*)*
- RMIX (safe) default: return *after* marshalling
  - more precisely: provider must ensure data consistency
  - may employ extra memory buffers and/or immutability detection
- Can be overridden on a stub-by-stub basis

Execution order

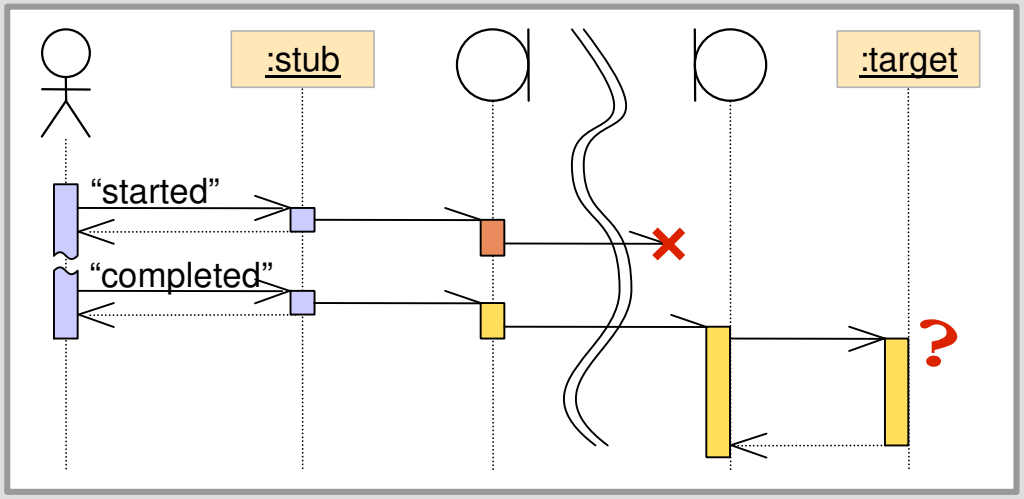
- RMI totally orders calls initiated from a single thread
- Obviously, this should be relaxed in asynchronous RMI
  - Yet, some ordering semantics are still practical and commonly expected – consider e.g. event notification:



- Solution in RMIX
  - Only calls made by thread *via the same stub* are ordered
  - (Somewhat similar to Microsoft RPC and ANSAware RPC)
  - To avoid ordering, client can always use distinct copies of stubs

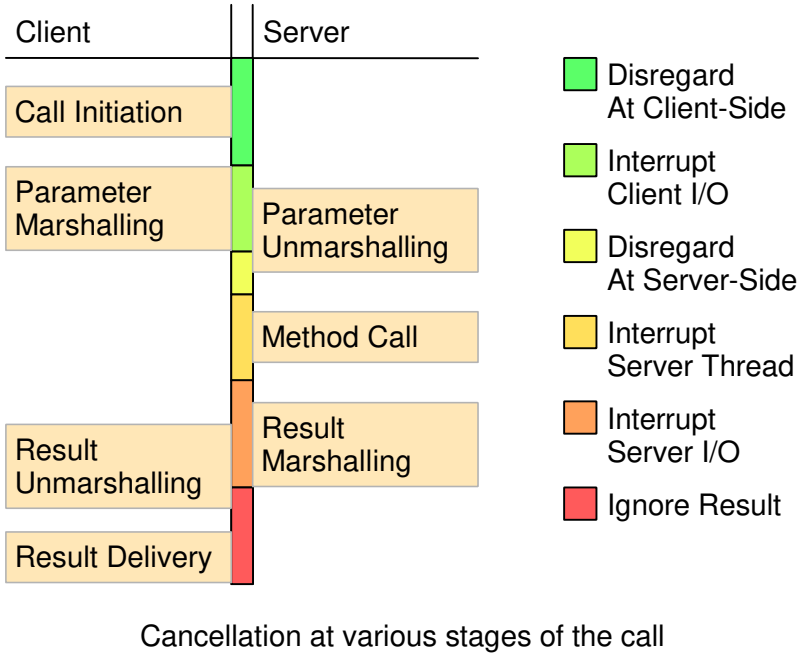
Exception handling

- Exceptions in asynchronous RMI are asynchronous
  - May arrive after further calls were initiated
  - Relying on ordering guarantees, methods may exhibit causality dependencies



- Solution in RMIX
  - Asynchronous exception state is *sticky* on a thread+stub pair (after failure, all subsequent calls fail automatically)
  - This feature can be disabled in a stub if undesired

Cancellation



Cancellation at various stages of the call

```
boolean cancel(
    boolean mayInterruptIfRunning);
```

- Inherently asynchronous
  - Requires different actions at different stages of the call
  - Will not always succeed
- Must be non-blocking
  - May not try to contact server before returning control
- Cancellation In RMIX
  - If failed, no side effects (as if it was never attempted)
  - Two modes supported: *conservative* and *best effort*
    - Conservative: successful only if the system can be put in the state as if the call was never initiated
    - Best effort: can report success and continue cancellation efforts in background

Security considerations

- Security control bound to threads of execution
  - Privileges depend on current invocation stack
  - “Protection Domain” and “AccessController” mechanisms
- Threads in asynchronous RMI
  - Background (system) threads perform some activities on behalf of the calling thread
  - In fact, they often execute user-level code, e.g. custom deserializers and callbacks
- Access control propagation
  - To avoid a vulnerability, RMIX provider must ensure proper context propagation from the calling thread
  - RMIX provides utility classes to facilitate implementations

Implementation status

- Asynchronous calls supported by all current RMIX communication providers
  - Asynchronous JRMPX, SOAP, ONC-RPC
  - Working and stable!
- Common, generic base class
  - Manages thread pooling, call ordering, exception handling policies, parameter consistency, and propagation of security context
  - Makes the specification relatively easy to implement (yet not completely painless)
- Yet, better throughput attainable by dedicated implementations
  - Depending on the protocol, they may be able maintain order at the server side rather than client side
  - High performance asynchronous ONC-RPC: work in progress

Related Work

- Several Async RMI implementations available
  - Employ futures, result queues
  - Some do not preserve server transparency
  - Some provide only one-way semantics
  - Focus on performance; no discussion of ordering, exception handling, cancellation, security
- ProActive: distributed framework with asynchronous calls
  - Precise semantics, but thread model somewhat different than traditional RMI
- CORBA Asynchronous Method Invocation
  - Callbacks and futures, synchronization modes, server transparency
  - No discussion of execution order, cancellation, security
- RPC systems
  - Selected aspects (ordering, failure handling) addressed by Microsoft RPC, ANSAware

Summary

- Asynchronous RMI – implicit increase of concurrency
- Main motivation: improved throughput
- Yet, raises subtle syntactic, semantic, and security issues
  - Lack of precisely defined semantics limits applicability
- RMIX: multiprotocol RMI framework
  - Features asynchronous calls with precise semantics, as described in this talk
  - Stable, publictly available, liberal open-source license
- Future work
  - High-performance provider implementations
  - Feasible for scientific message passing applications

Thank You!