RMIX: A Multiprotocol RMI Framework for Java

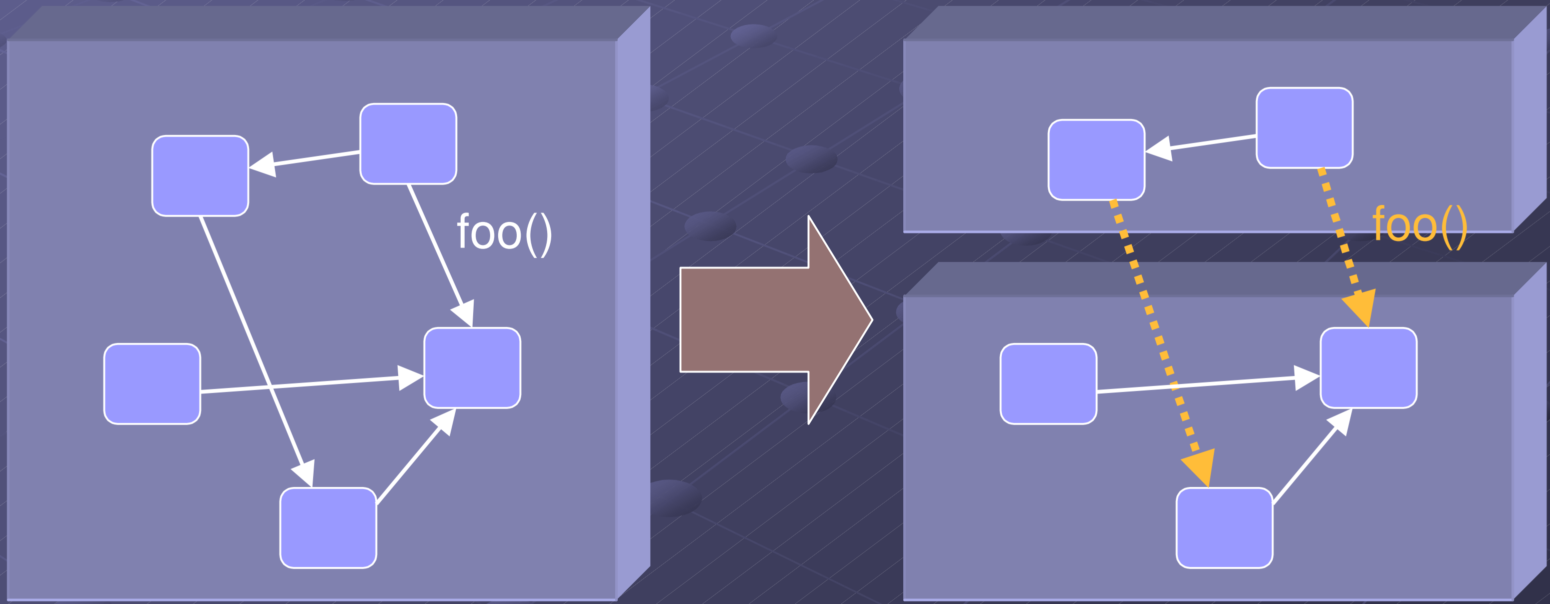
D. Kurzyniec, T. Wrosek, A. Slominski, V. Sunderam

Emory University, Indiana University

Research supported by U.S. DoE MICS and NSF ACR

RMI: Java-based Distributed Computing

- Java object model extended for multiple, distributed VMs
- Simplicity, easy to distribute existing code
- RPC paradigm with the benefits of the object model
  - serializable remote references, callbacks, dynamic export, ...



Issue #1: Interoperability

- Java Remote Method Protocol (JRMP)
  - Sophisticated and highly Java-specific
  - Disallows connectivity with non-Java services
- Alternatives: RMI-IIOP, JAX-RPC, ...
  - Independent solutions using different wire protocols (IIOP, SOAP, ...)
  - Differences in the programming model
  - Effect: difficult to port code between libraries
  - And what about multiprotocol applications?...

Issue #2: Performance

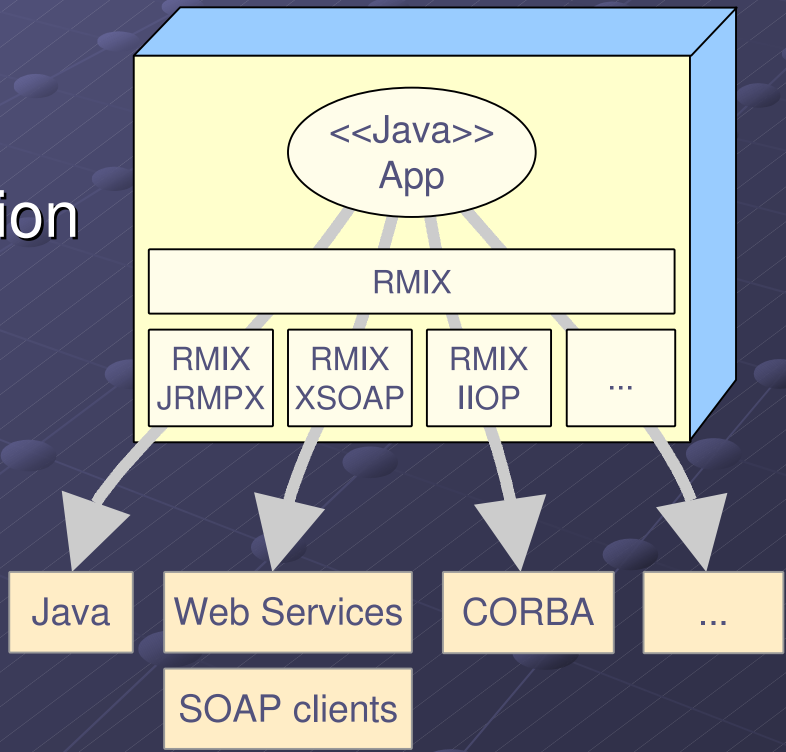
- Early implementations of RMI (Java 1.1)
  - Poor communication performance
  - Not targeted for high-bandwidth networks
- Current status (2003, Java 1.4)
  - Significant improvements, especially for primitive types and arrays thereof
  - However, impossible to optimize object serialization without breaking compatibility
- Solutions
  - Alternative RMI implementations (KaRMI, Manta, ...)
  - Not interoperable, useful only in tightly-coupled applications

Issue #3: Stiff programming model

- Security in loosely-coupled systems
  - Lack of primitives enabling authentication (e.g. ability to create independent server references to the same object for different client sessions)
  - No message-level security features (encryption, MAC, replay detection)
- Communication paradigm
  - No support for asynchronous or one-way calls
  - Some attempts: ARMI, ...

RMIX – design goals

- Enable various RMI solutions within a single, provider-based framework library
- Enable high-performance communication
- Enable interoperability with heterogeneous services
- Offer more flexible communication model, but retain simplicity
- Easy deployment



RMIX – the programming model

- Object model inherited from Java RMI
- Remote proxies
- Remote interfaces
- Remote methods
- Remote exceptions
- Parameters passed by value

RMIX – „Hello, World!”

```
public interface Hello
       extends java.rmi.Remote
{
  String hello(String msg)
    throws java.rmi.RemoteException;
}
```

Hello.java

```
import edu.emory.mathcs.rmix.Naming;

public class HelloSrv implements Hello
{
  public String hello(String msg) {
    return ”Hello, ” + msg;
  }

  public static void main(String[] args)
  {
    HelloSrv hello = new HelloSrv();
    Naming.rebind(”hello”, hello);
  }
}
```

HelloSrv.java

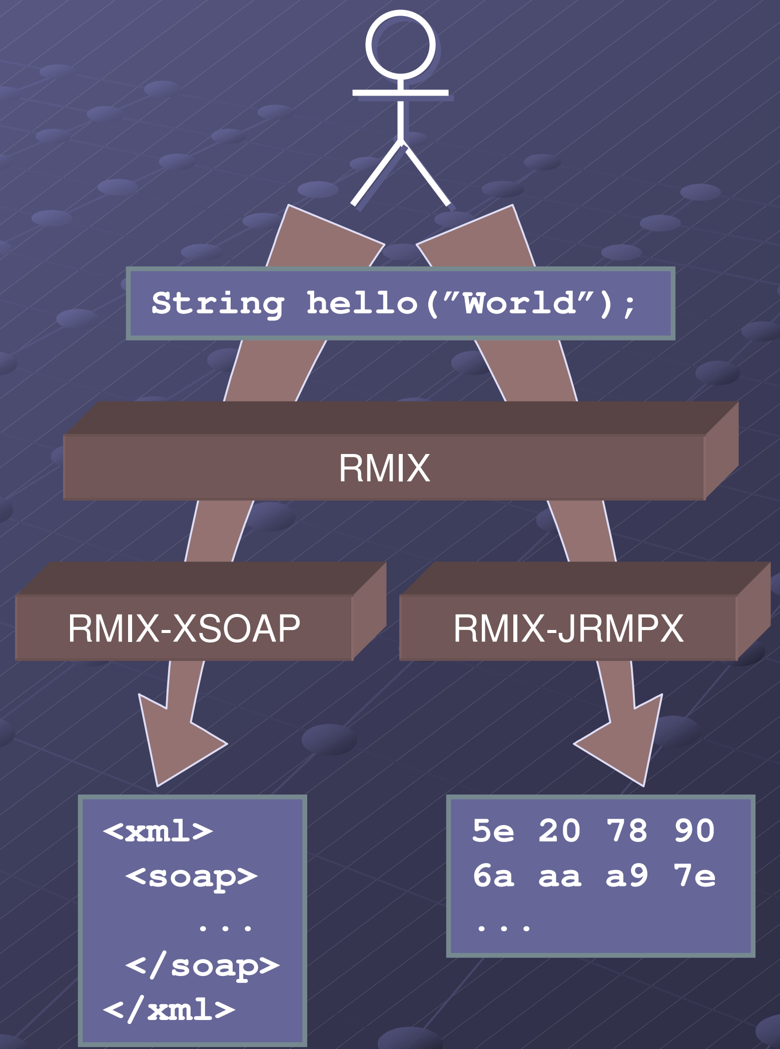
```
import edu.emory.mathcs.rmix.Naming;

public class HelloClient
{
  public static void main(String[] args)
  {
    String srv = args[0];
    String name = ”//” + srv + ”/hello”;
    Hello hello =
      (Hello)Naming.lookup(name);
    String response = hello.hello(”World”);
    System.out.println(response);
  }
}
```

HelloClient.java

RMIX – data marshalling

- Remote transmission
  - Calls, their parameters, return values, and exceptions are sent over the network
  - Serial form depends on the protocol
  - Marshalling by pluggable provider modules
- Framework defines only the *semantics*
  - Serializable types of parameters, return values and exceptions
  - What the application may depend on?
  - What the provider modules must support?



RMIX – data marshalling (cont.)

- Goal: accomodate even very simple protocols
  - Examples: Sun-RPC / XDR
  - Limited number of serializable types
  - No distributed garbage collection, polymorphism, object graphs
- Minimal required semantics must then be very narrow
  - Primitive types, strings, arrays thereof
  - Bean-like classes
  - Support for more sophisticated types is the provider „Quality of Implementation"
- Runtime „capability query" mechanisms considered
  - Accomodate applications that require more than bare minimum, but without binding to a specific provider
  - Let protocol providers advertise their marshalling capabilities
  - The application may then query RMIX for providers that fulfill given requirements

RMIX – protocol switching

- Goal: program can dynamically switch communication protocol
  - Run-time protocol negotiation
- Prerequisite: any provider must be able to marshall stubs of any other provider
  - But we do not want inter-provider dependencies!

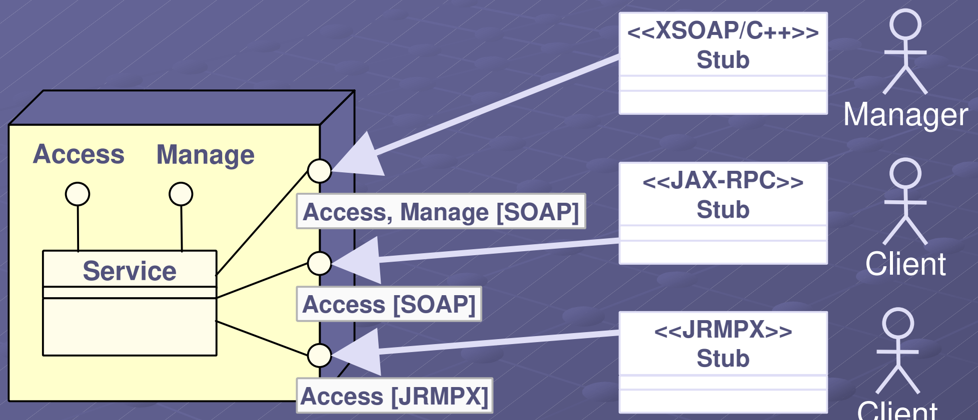
```
public class HelloImpl
       implements Hello
{
  Hello switch(String newProt) {
    Hello newStub = (Hello)
     Rmix.export(newProt, this)
     .bind();
    return newStub;
  }
}
```

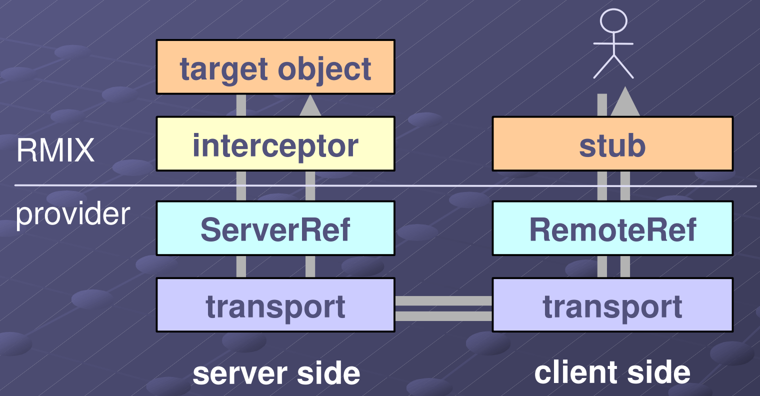
- Solution: easily serializable *unified reference format*
  - Every provider is able to unify (pre-serialize) its own stubs
  - Then, providers need to deal only with unified stubs

RMIX – Naming service

- Variety of bootstrap discovery mechanisms in use
  - rmiregistry, UDDI, LDAP, ...
  - Even out-of bounds techniques (like e-mail) are possible
  - In systems with distributed garbage collection (DGC), only system-specific registries can guarantee *liveness* of stored references
- RMIX leaves the choice to the application
  - Unified remote reference format is easily serializable and can be stored in any naming service
- Backward compatibility with rmiregistry
  - All RMIX stubs can be stored in the rmiregistry
  - For providers supporting DGC, stored references are live

RMIX enhanced export semantics





- Multiple endpoints per object
  - E.g. for different users, with different security characteristics
- Possible to expose only a subset of remote interfaces
- Remote objects may have *invocation interceptors*
  - Access policy may be separated from the application

RMIX – other features

- Non-blocking communication modes (experimental)
  - Asynchronous and one-way calls, using *futures* and *callbacks*
  - Full description is out of scope
- Message security Quality-of-Service (experimental)
  - Data encryption, MAC, replay detection (can be independently turned on and off)
  - Can be used with any RMI implementation based on socket factories, including standard RMI
  - Full description is out of scope
- Dynamic stubs
  - RMIX stubs are created dynamically – rmic is not needed
  - Simplified application deployment and maintenance
- Provider-specific controls
  - *Parameters API* provides access to provider-specific settings

RMIX deployment

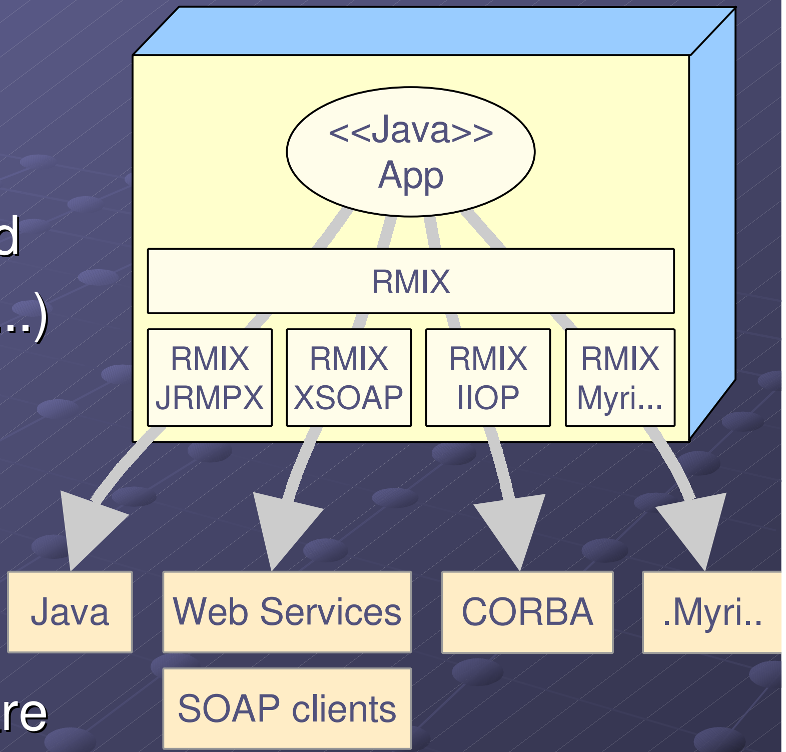
- Unified provider binary format
  - Every provider module is a JAR file with the appropriate metadata in the manifest
  - Installation of provider modules = putting JARs in the filesystem
- Dynamic discovery
  - Application can query RMIX for available provider modules
  - Newly added JAR files are immediately visible

```
<dir>/
  rmix/
    lib/
      rmix.jar
    provider/
      rmix-xsoap.jar
      rmix-jrmpx.jar
      ...
```

```
java
  -Xbootclasspath/a:<dir>/rmix/lib/rmix.jar
  -Drmix.home=<dir>/rmix/ ...
```

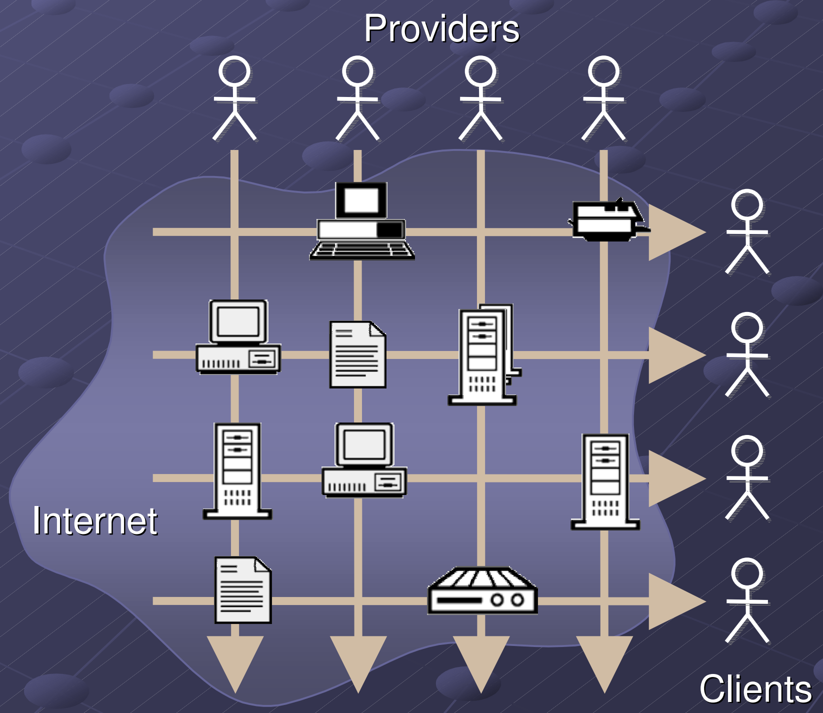
Supplied provider modules

- RMIX-JRMPX
  - Full-featured replacement of standard RMI (Java ser, DGC, object graphs, ...)
  - Lightweight, tunnels calls through standard RMI
  - Performance close to standard RMI
- RMIX-XSOAP
  - Based on standalone XSOAP software
  - Use SOAP 1.1 as a wire protocol
  - Enable Web Service connectivity and hosting in J2SE



Application example – H2O

- Distributed resource sharing framework
  - *Providers* independently share *resources* over the network
  - *Clients* utilize resources by uploading application components *(pluglets)* into them
  - RMIX used as a communication substrate
- Dynamic protocol negotiation
  - SOAP used as a first-contact protocol
  - Application modules may be bound with more efficient protocols
- Multi-user environment
  - Client authentication, secure sessions



Migration to RMIX

- Motivation
  - Simplify deployment using dynamic stubs
  - Port the application to a different communication protocol
- General approach
  - Usually, only mechanical changes at the server side needed:

`class A extends java.rmi.server.UnicastRemoteObject`	`class A extends edu.emory.mathcs.rmix.server.UnicastRemoteObject`
`java.rmi.Naming.rebind(obj);`	`java.rmi.Naming.rebind(Rmix.wrap(obj));`

  - In RMI, rmiregistry needs to access stub and interface classes; RMIX is able to overcome that:

`java.rmi.Naming.rebind(obj);`	`edu.emory.mathcs.rmix.Naming.rebind(obj);`
`java.rmi.Naming.lookup(obj);`	`edu.emory.mathcs.rmix.Naming.lookup(obj);`

Migration – problematic cases

- Some cases require manual changes in the code
- Explicit dependencies on the RMI class inheritance hierarchy
  - E.g. „`obj instanceof java.rmi.server.RemoteObject`"
  - RMIX tries to mimic RMI class hierarchy to mimimize problems
- Serialization of remote objects that were not exported
  - In RMI, such objects are passed by value
  - RMIX performs automatic export and returns a stub
  - However, relying on the RMI semantics (introduced in Java 1.3) is questionable practice at best

Summary

- RMIX – comprehensive, complete extension of Java RMI
  - Dynamic stubs, asynchronous calls, security, pluggable protocols behind unified API, dynamic protocol deployment, export enhancements
- Experiences
  - Successfully applied in the H2O metacomputing system
  - Loosely-coupled, multi-user, multi-protocol environment
  - Dynamic protocol negotiation
- Ongoing and future work
  - High-performance transport providers
  - Stronger application independence of communication protocols