Java RMI, Protocol Buffer & Pub-sub

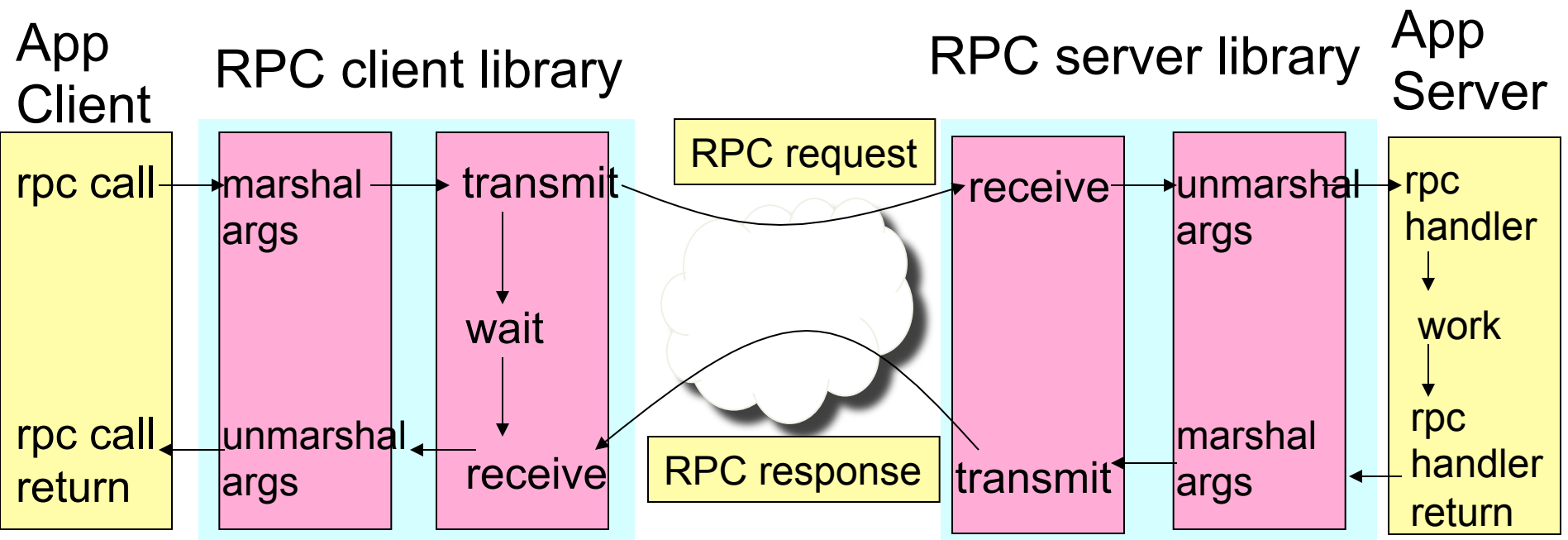
Haibo Chen

Some slides adapted from Bryan Carpenter and Vinay Kolar

Outline

- Recap
- Java RMI
- Google Protocol Buffer
- Google Publish-Subscribe

Recap: RPC architecture



Recap: LRPC Concepts

Execution model of LRPC is borrowed from "protected procedure call" of capability systems

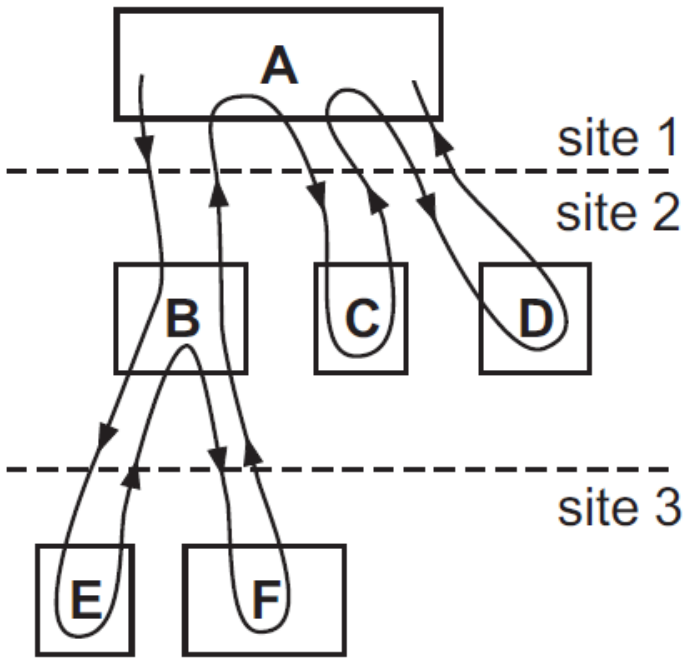
Programming semantics and large-grained protection model of LRPC are borrowed from RPC

Recap: Four techniques of LRPC

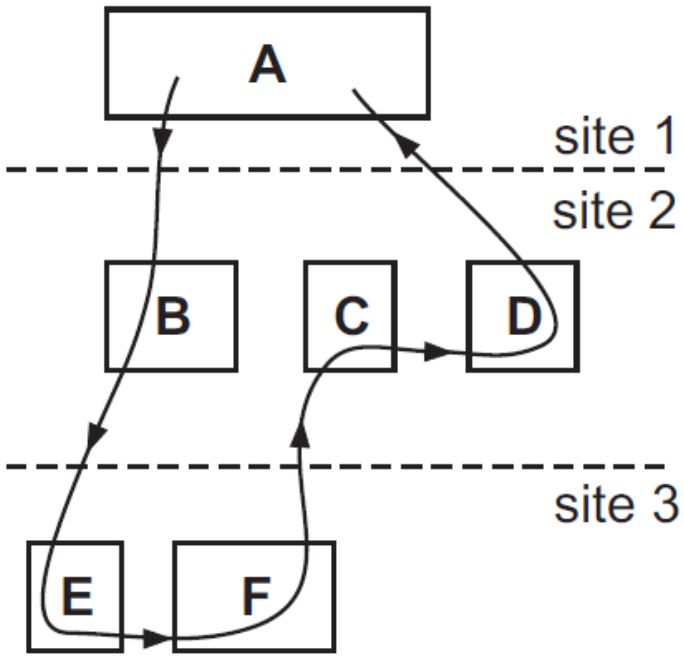
- Simple control transfer
  - client's thread executes the requested procedure in server's domain
- Simple data transfer
  - Parameter-passing mechanism similar to procedure call
  - Shared argument stack eliminates redundant data copying
- Simple stubs
  - generation of highly-optimized stubs
- Design for concurrency
  - Avoids shared data structure bottlenecks
  - Benefits from speed-up of multiprocessor

Recap: RPC Chain

Standard RPC



RPC Chain



Recap: Chaining Function

```
// service function
object sf(object parmlist)
  // parmlist:  parameter list

// chaining function
nexthop cf(object state, object result)
  // state:  from client or earlier parts of chain
  // result:  from last preceding service function
  // returns next chain hop:
  //         (server, sf_name, parmlist,
  //         cf_name, state)

---

chain_id start_chain(machine_t server,
        string sf_name, object parmlist,
        string cf_name, object state)
```

① call sf()

② call cf()

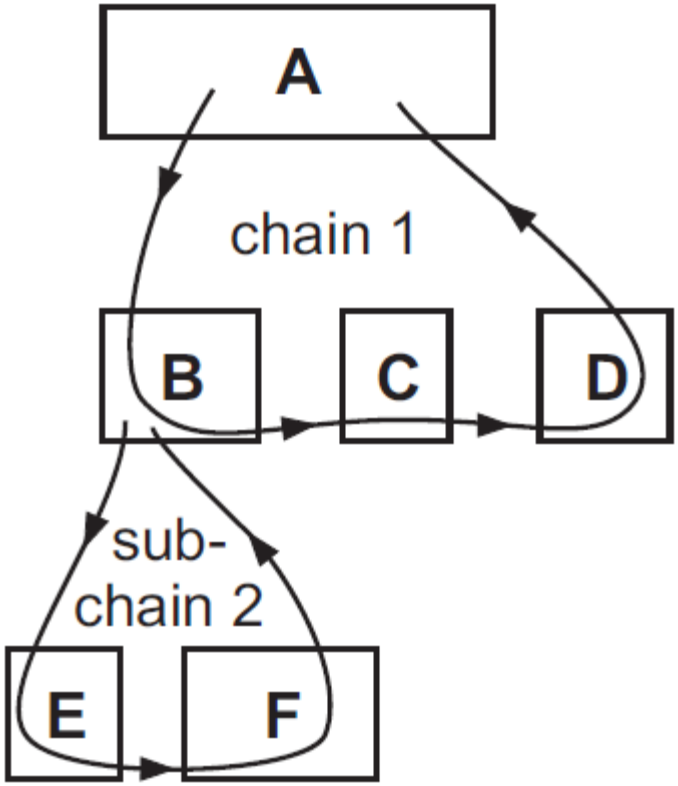
Reformat / chose hop

③ return start_chain

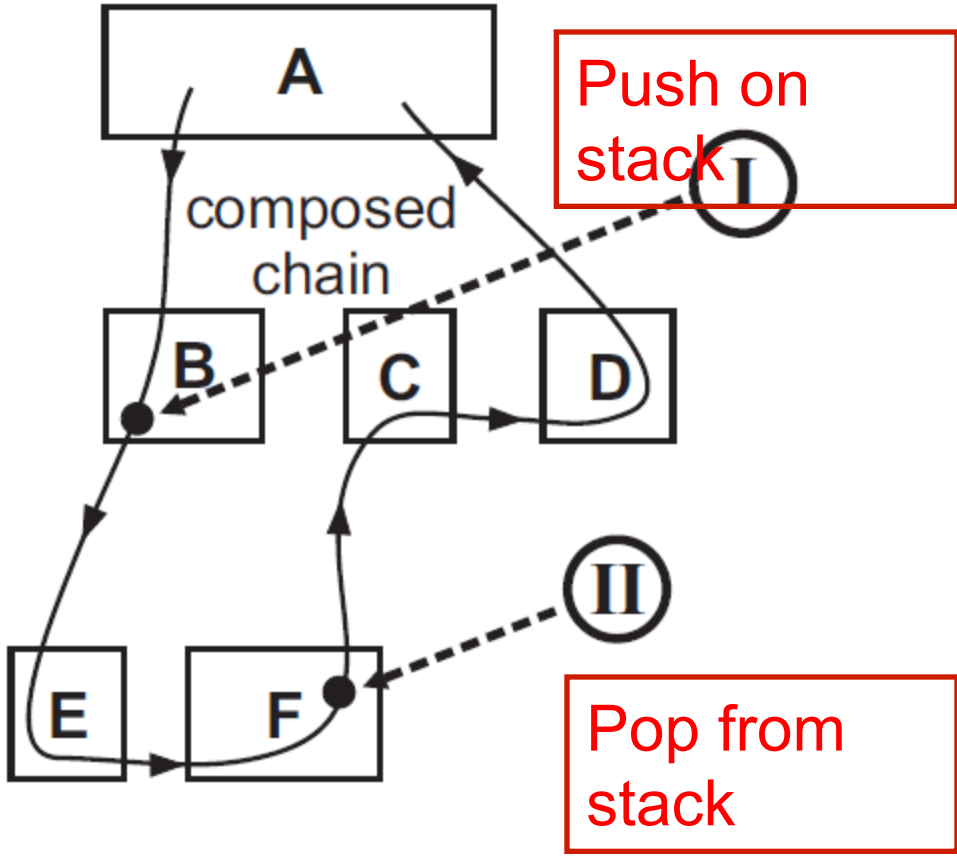
Figure 2: **(Top)** Signature of a service function (sf) and chaining function (cf). **(Bottom)** Signature of function that launches an RPC chain.

Recap: Nested and Composition chains

Nested



Composition



Examples and Evaluation

Storage applications

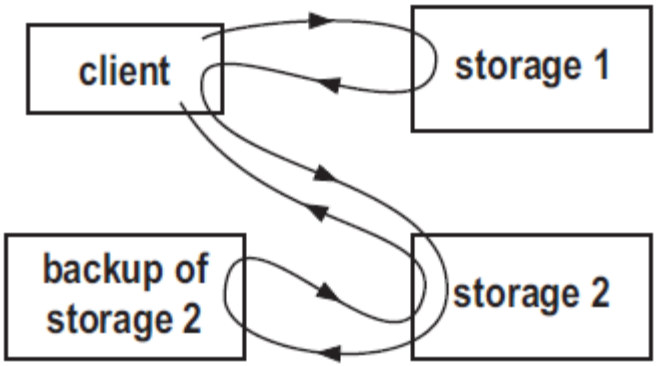
Web mail application

Micro-benchmarks

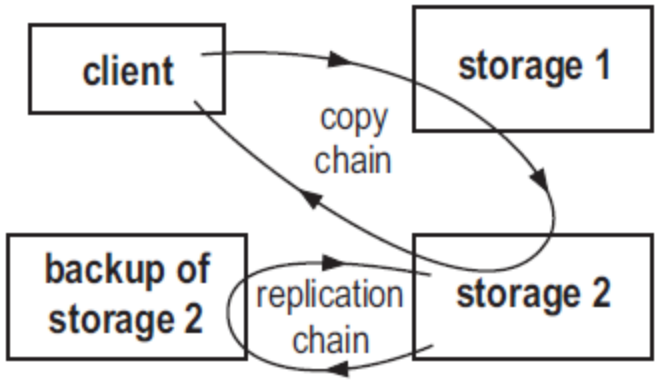
Application 1: Storage

Copy data from S1 to S2 with a backup server

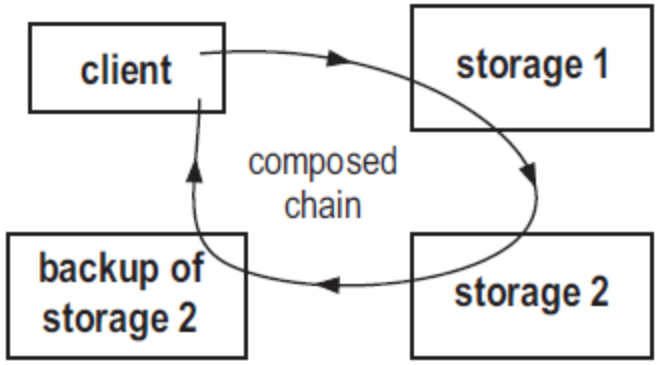
(a)



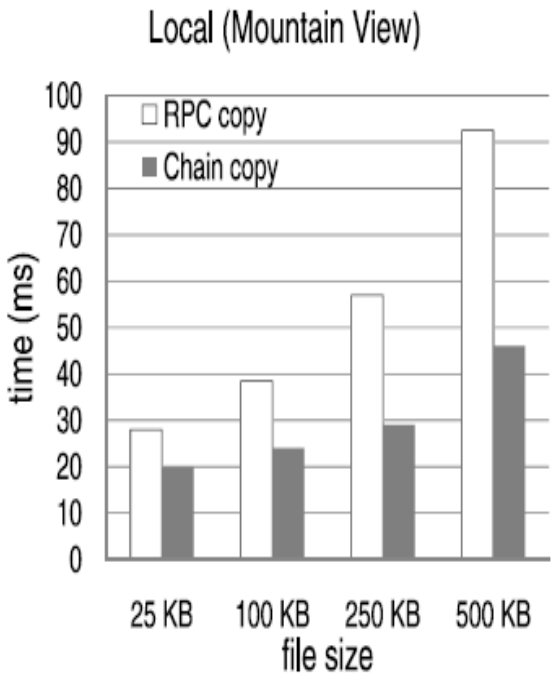
(b)

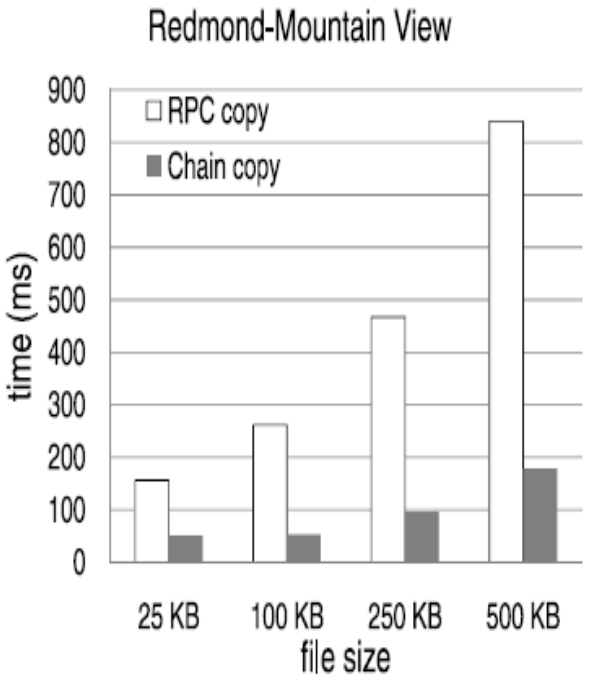


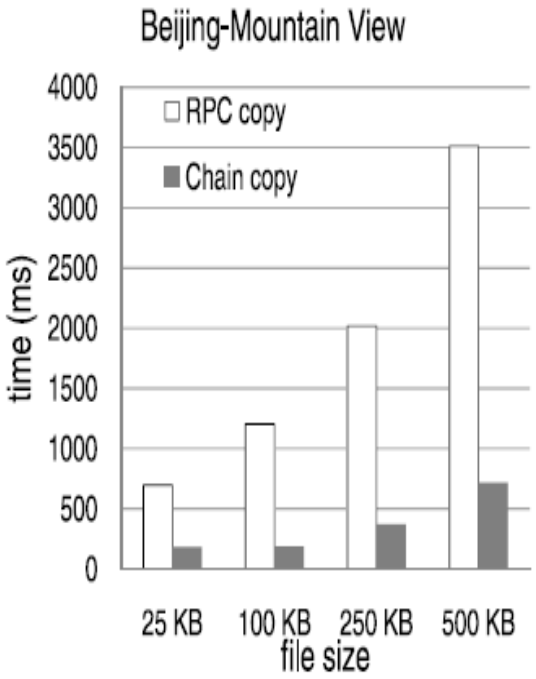
(c)



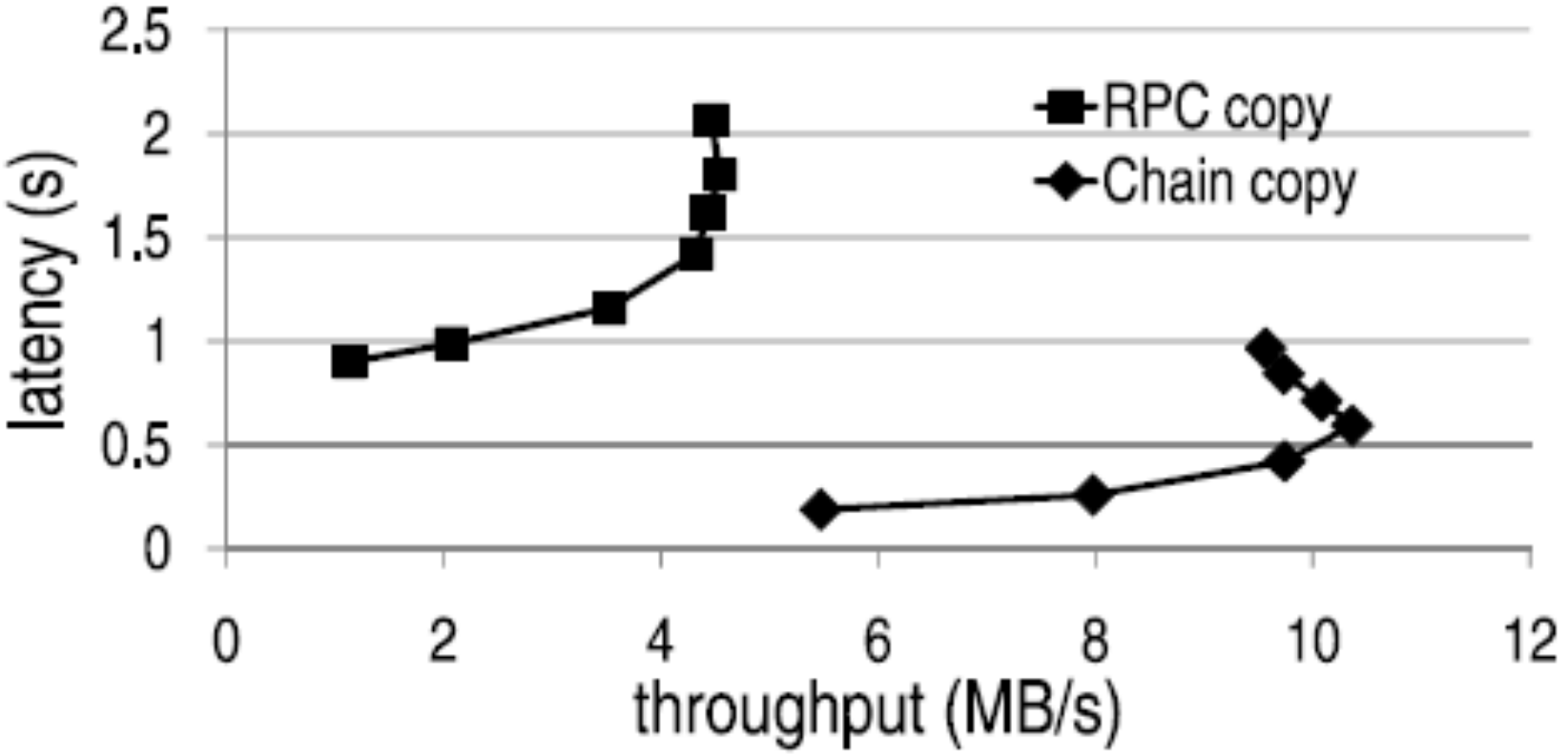
Copy Performance



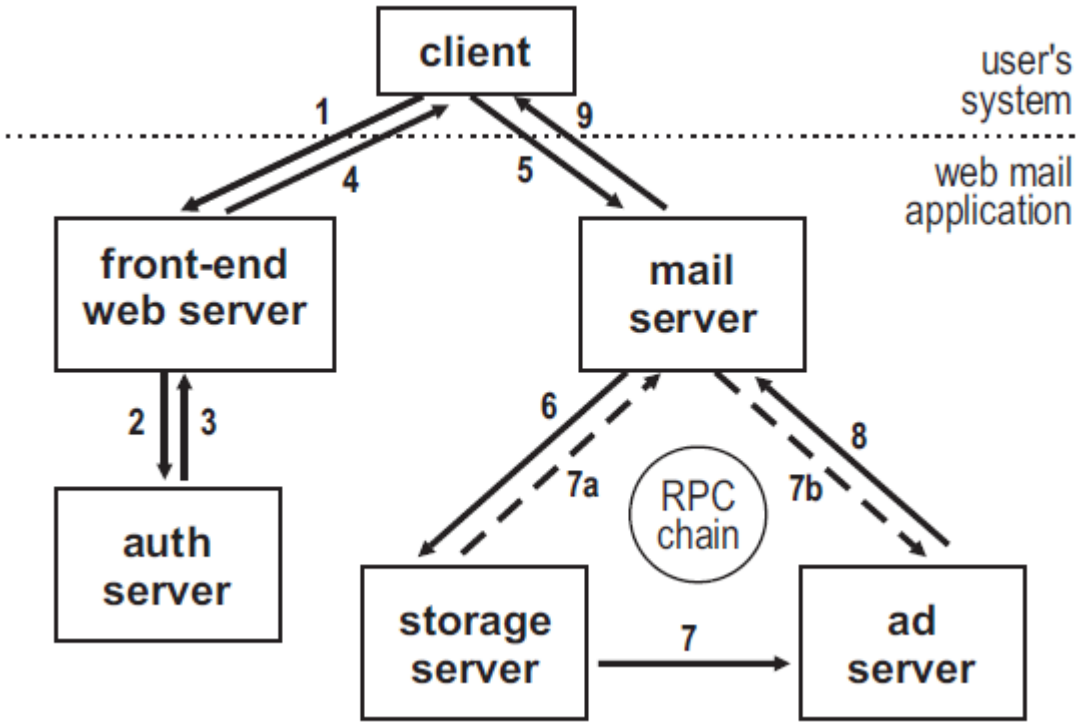


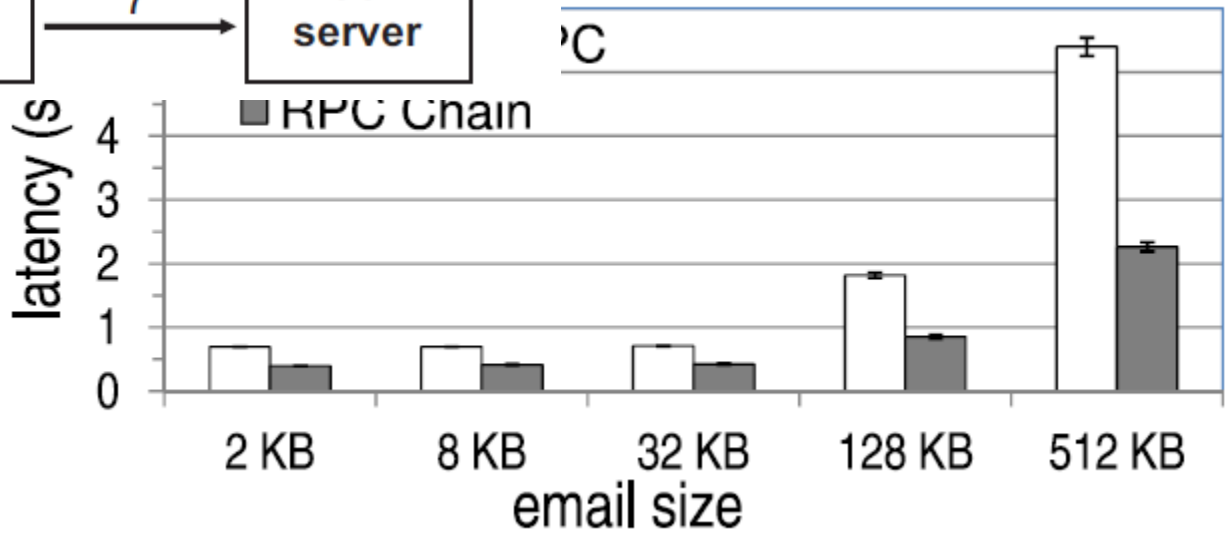


Copy Performance



Application 2: Web Mail





Micro-benchmarks

- Overhead of chaining function(compile time)
- State transfer overhead

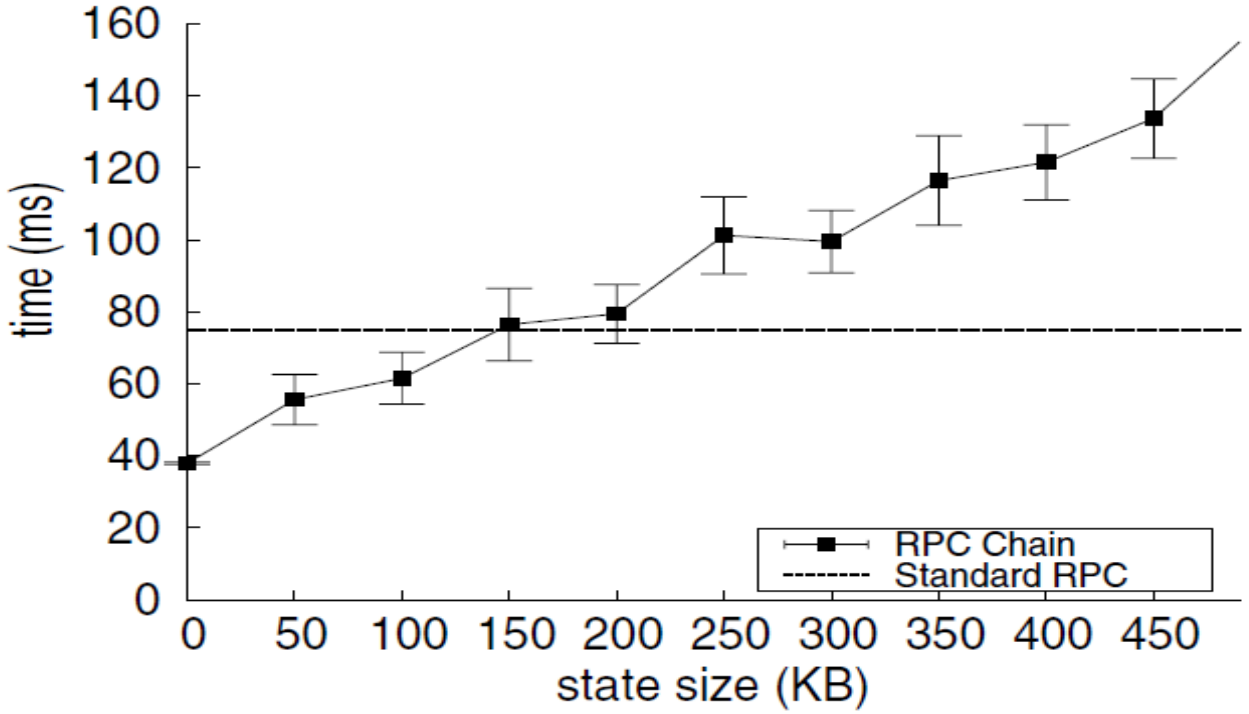


Figure 10: Execution time using an RPC chain versus standard RPC to call 2 servers.

References

"Lightweight Remote Procedure Call" by B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy, Proceedings of the 12th Symposium on Operating Systems Principles, pp. 102-113, December 1989

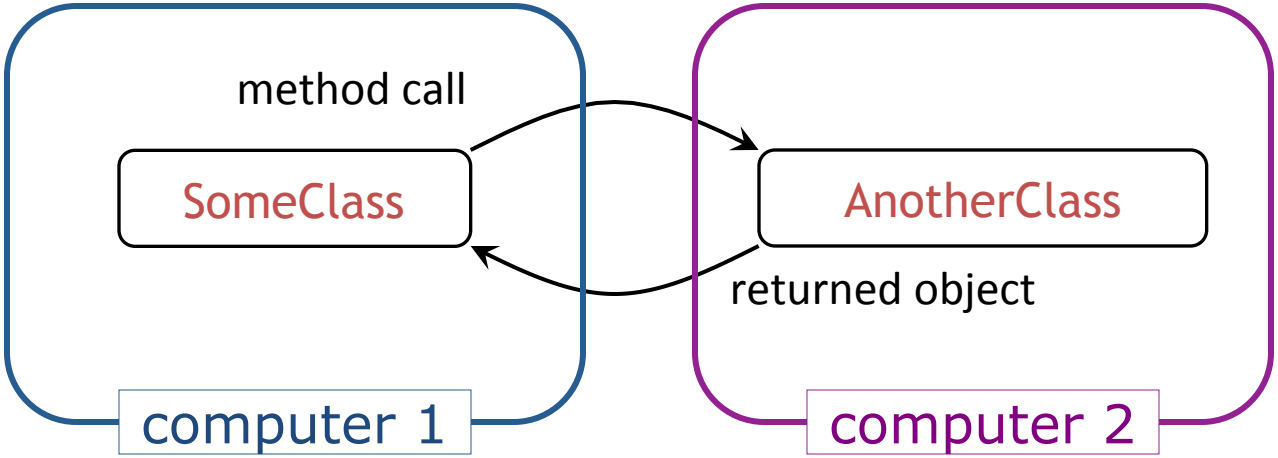
"Implementing Remote Procedure Calls" by A. D. Birrell and B. J. Nelson, ACM Transactions on Computer Systems, Vol. 2, No. 1, pp. 39-59, February 1984

“rpc chains efficient client-server communication in geo distributed systems” by Yee Jiun Song, Marcos K. Aguilera, Ramakrishna Kotla, Dahlia Malkhi. In Proceedings of NSDI, 2009.

Java RMI

"The network is the computer"

- Consider the following program organization:

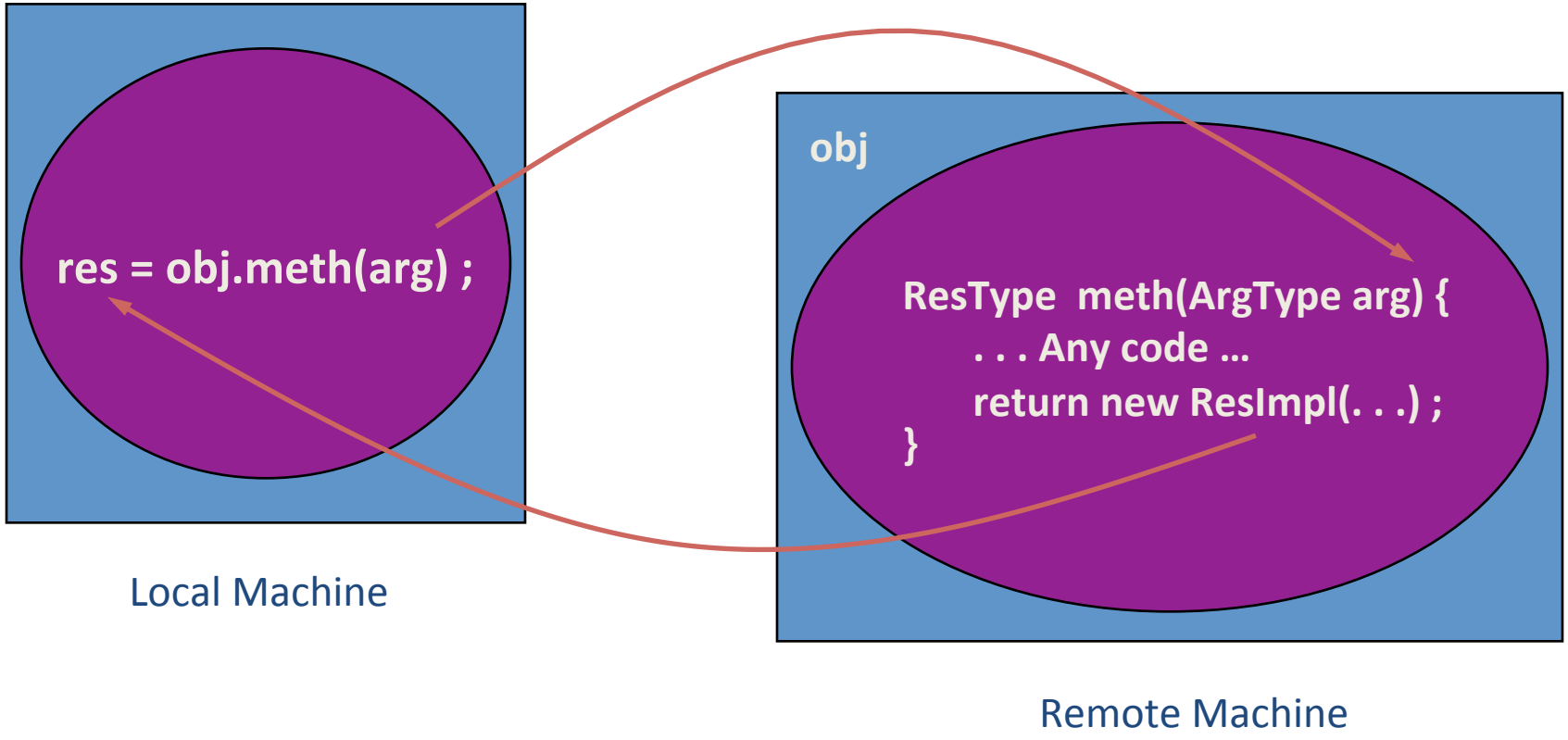


- If the network *is* the computer, we ought to be able to put the two classes on different computers

- RMI is one technology that makes this possible

Distributed Object Picture

Code running in the local machine holds a *remote reference* to an object **obj** on a remote machine:



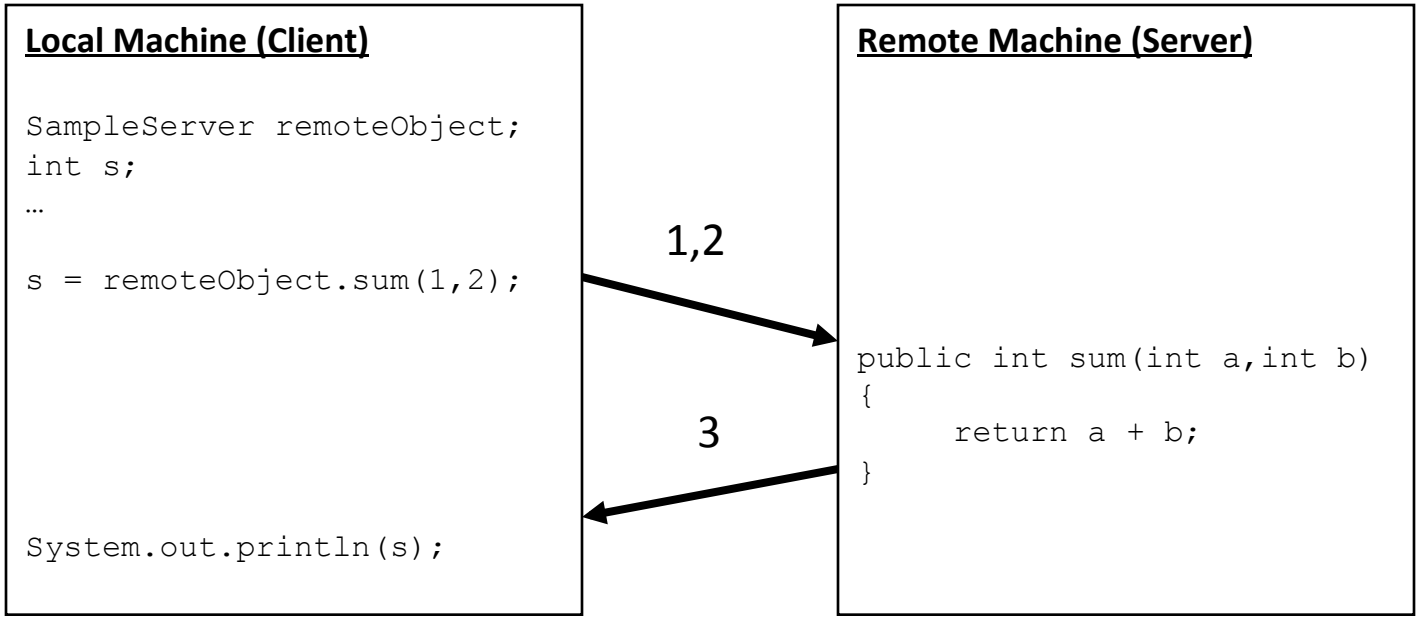
RMI and other technologies

- CORBA (Common Object Request Broker Architecture) was used for a long time
  - CORBA supports object transmission between virtually any languages
  - Objects have to be described in IDL (Interface Definition Language), which looks a lot like C++ data definitions
  - CORBA is complex and flaky
  - CORBA has fallen out of favor
- Microsoft support CORBA, then COM, now .NET

- RMI is purely Java-specific
  - Java to Java communications only
  - As a result, RMI is much simpler than CORBA

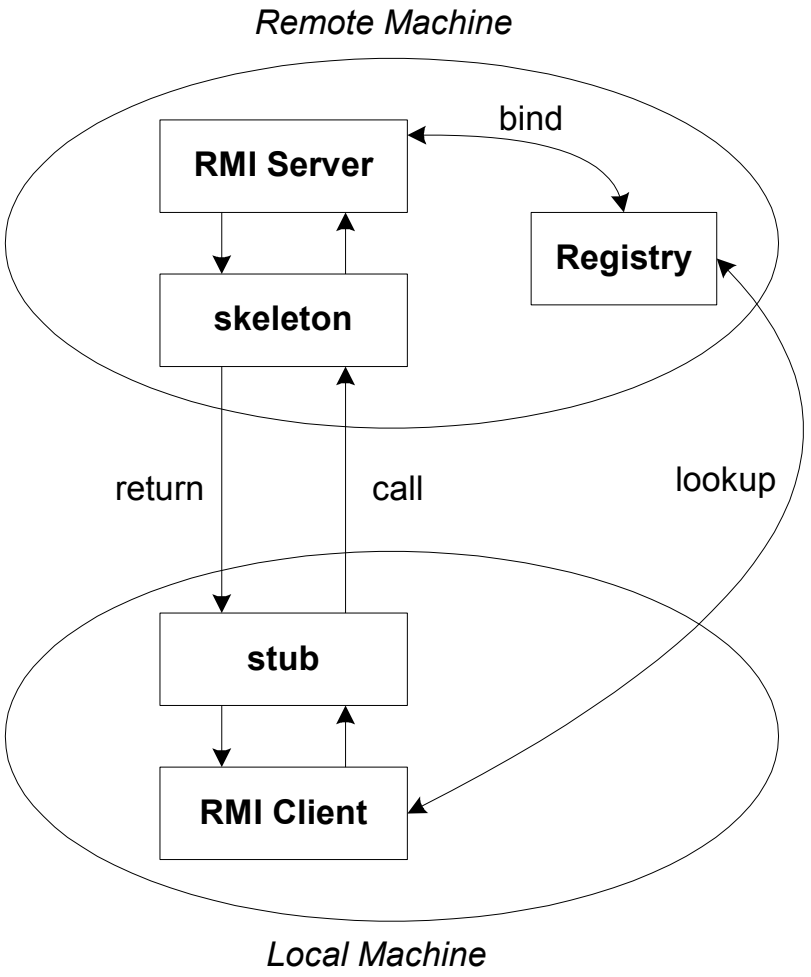
Java Remote Object Invocation (RMI)

- Overview of RMI
- Java RMI allowed programmer to execute remote function class using the same semantics as local functions calls.

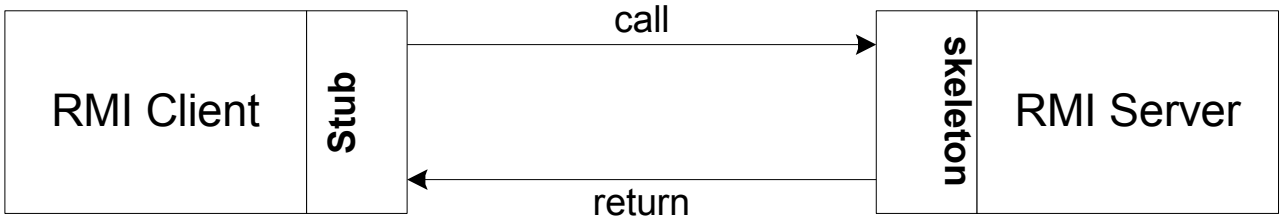


The General RMI Architecture

- The server must first bind its name to the registry
- The client lookup the server name in the registry to establish remote references.
- The Stub serializing the parameters to skeleton, the skeleton invoking the remote method and serializing the result back to the stub.



The Stub and Skeleton



- A client invokes a remote method, the call is first forwarded to stub.
- The stub is responsible for sending the remote call over to the server-side skeleton
- The stub opening a socket to remote server, marshaling parameters and forwarding data stream to skeleton.
- A skeleton contains a method that receives remote calls, unmarshals parameters, and invokes actual remote object implementation.

What is needed for RMI

- Java makes RMI (Remote Method Invocation) *fairly* easy, but there are some extra steps

- To send a message to a remote "server object,"
  - The "client object" has to *find* the object
    - Do this by looking it up in a registry
  - The client object then has to marshal the parameters (prepare them for transmission)
    - Java requires Serializable parameters
    - The server object has to unmarshal its parameters, do its computation, and marshal its response
  - The client object has to unmarshal the response

- Much of this is done for you by special software

Is RMI a Language Extension?

- The internal mechanics of remote invocation are much more complex than local invocation:
  - Marshal arguments
    - collected together into messages suitable for shipping across the Internet
  - Marshal results (or exceptions) are similarly shipped back

- Perhaps surprisingly, RMI involves *no* modification to the Java language, compiler, or virtual machine.
  - The illusion of remote invocation is achieved by clever libraries, plus one relatively simple "post-processor" tool (**rmic**).

Processes

- For RMI, you need to be running *three* processes
  - The Client
  - The Server
  - The Object Registry, rmiregistry, which is like a DNS service for objects
- You also need TCP/IP active

Interfaces

- Interfaces define behavior
- Classes define implementation

- Therefore,
  - In order to use a remote object, the client must know its behavior (interface), but does not need to know its implementation (class)
  - In order to provide an object, the server must know both its interface (behavior) and its class (implementation)
- In short,
  - The interface must be available to both client and server
  - The class of any transmitted object must be on both client and server
  - The class whose method is being used should only be on the server

Classes

- A Remote class is one whose instances can be accessed remotely
  - On the computer where it is defined, instances of this class can be accessed just like any other object
  - On other computers, the remote object can be accessed via object handles

- A Serializable class is one whose instances can be marshaled (turned into a linear sequence of bits)
  - Serializable objects can be transmitted from one computer to another

Conditions for serializability

- If an object is to be serialized:
  - The class must be declared as public
  - The class must implement Serializable
    - However, Serializable does not declare any methods
  - The class must have a no-argument constructor
  - All fields of the class must be serializable: either primitive types or Serializable objects
    - Exception: Fields marked transient will be ignored during serialization

Remote interfaces and class

- A Remote class has two parts:
  - The interface (used by both client and server):
    - Must be public
    - Must extend the interface java.rmi.Remote
    - Every method in the interface must declare that it throws java.rmi.RemoteException (other exceptions may also be thrown)
  - The class itself (used only by the server):
    - Must implement the Remote interface
    - Should extend java.rmi.server.UnicastRemoteObject
    - May have locally accessible methods that are not in its Remote interface

Security

It isn't safe for the client to use somebody else's code on some random server

```
System.setSecurityManager(new RMISecurityManager());
```

The security policy of `RMISecurityManager` is the same as that of the default `SecurityManager`

Your client program should use a more conservative security manager than the default

The server class

- The class that defines the server object should extend UnicastRemoteObject
  - This makes a connection with exactly one other computer
  - If you must extend some other class, you can use exportObject() instead
  - Sun does *not* provide a MulticastRemoteObject class
- The server class needs to register its server object:
  - String url = "rmi://" + ***host*** + ":" + ***port*** + "/" + ***objectName***;
    - The default port is 1099
  - Naming.rebind(url, ***object***);
- Every remotely available method must throw a RemoteException (because connections can fail)
- Every remotely available method should be synchronized

RMI Server: interface

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RmiServerIntf extends Remote {
    public String getMessage() throws RemoteException;
}
```

RMI Server: class

```
public class RmiServer extends UnicastRemoteObject
    implements RmiServerIntf {
    public static final String MESSAGE = "Hello world";

    public String getMessage() {
        return MESSAGE;
    }

    public static void main(String args[]) {
        System.out.println("RMI server started");
        System.setProperty("java.security.policy","file:./rmi.policy");

        // Create and install a security manager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
            System.out.println("Security manager installed.");
        } else {
            System.out.println("Security manager already exists.");
        }
```

```
try { //special exception handler for registry creation
        LocateRegistry.createRegistry(10999);
        System.out.println("java RMI registry created.");
    } catch (RemoteException e) {
        //do nothing, error means registry already exists
        System.out.println("java RMI registry already exists.");
    }

    try {
        //Instantiate RmiServer
        RmiServer obj = new RmiServer();

        // Bind this object instance to the name "RmiServer"
        Naming.rebind("//localhost:10999/RmiServer", obj);

        System.out.println("PeerServer bound in registry");
    } catch (Exception e) {
        System.err.println("RMI server exception:" + e);
        e.printStackTrace();
    }
  }
}
```

RmiClient

```
public class RmiClient {
    // "obj" is the reference of the remote object
    RmiServerIntf obj = null;

    public String getMessage() {
        try {
            obj = (RmiServerIntf)Naming.lookup("//localhost:10999/RmiServer");
            return obj.getMessage();
        } catch (Exception e) {
            System.err.println("RmiClient exception: " + e);
            e.printStackTrace();

            return e.getMessage();
        }
    }
```

RmiClient

```
public static void main(String args[]) {
        // Create and install a security manager
       System.setProperty("java.security.policy","file:./rmi.policy");
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }

        RmiClient cli = new RmiClient();

        System.out.println(cli.getMessage());
    }
}
```

rmic

The class that implements the remote object should be compiled as usual

Then, it should be compiled with rmic:

```
rmic RmiServer
```

This will generate file RmiServer_Stub.class

This class do the actual communication

The “Stub” class must be *copied* to the client area

Garbage Collection

Remote objects are also garbage collected as follows:

- A *remote reference layer* on the server keeps a reference count for each locally held remote object implementation.
- A remote reference layer on the client notifies the server when its locally held references to the object are no longer in use.
- When all references from all clients have gone (i.e. when the reference count is zero), the server object is garbage collected

What if a client fails to notify the server?

- A client with a remote reference must periodically renew a lease with the run-time system on the server
- If the client holding the reference exits prematurely (without notifying the server)
- it will also stop renewing its leases. If a lease is not renewed on time, the server assumes the client has died, and the reference count is decremented anyway

Protocol Buffer

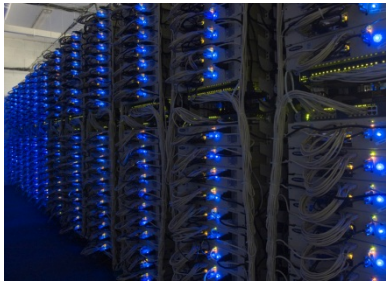
Google Case Study

- Google provides a fascinating case study of Distributed Systems
  - Scalability:
    - Google has scaled from an initial production system in 1998 to handling 88 billion queries a month
  - Reliability and Fault-tolerance:
    - The main search engine has never experienced an outage.
    - Users can expect a query result in 0.2 seconds
      - 24x7 availibility with 99.9% Service Level Agreement for paying customers
  - Variety of software applications are supported on Google Infrastructure
    - Google has built a generic infrastructure that handles many varying web applications (search, maps, voice, email, social networking, ads, docs, …)
  - Google is offering Platform As A Service
    - With the launch of Google App Engine, Google is stepping beyond providing software services
    - It is now providing its distributed systems infrastructure for application developers

Physical Model of a Google DS

- Google has created a large distributed system from commodity PCs

Commodity PC



Rack

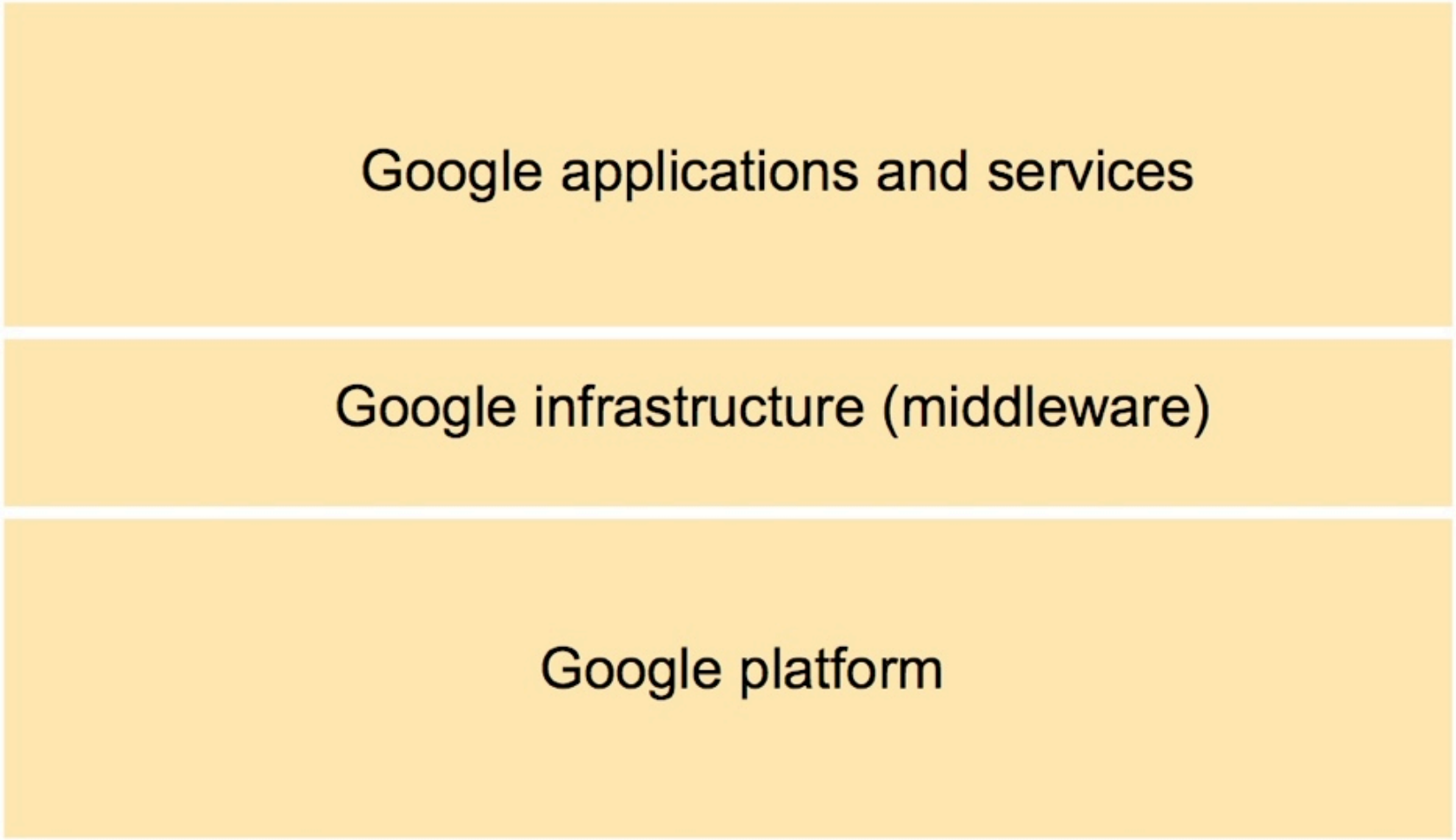
Approx 40 to 80 PCs
One Ethernet switch (Internal=100Mbps, external = 1Gbps)

Data Center

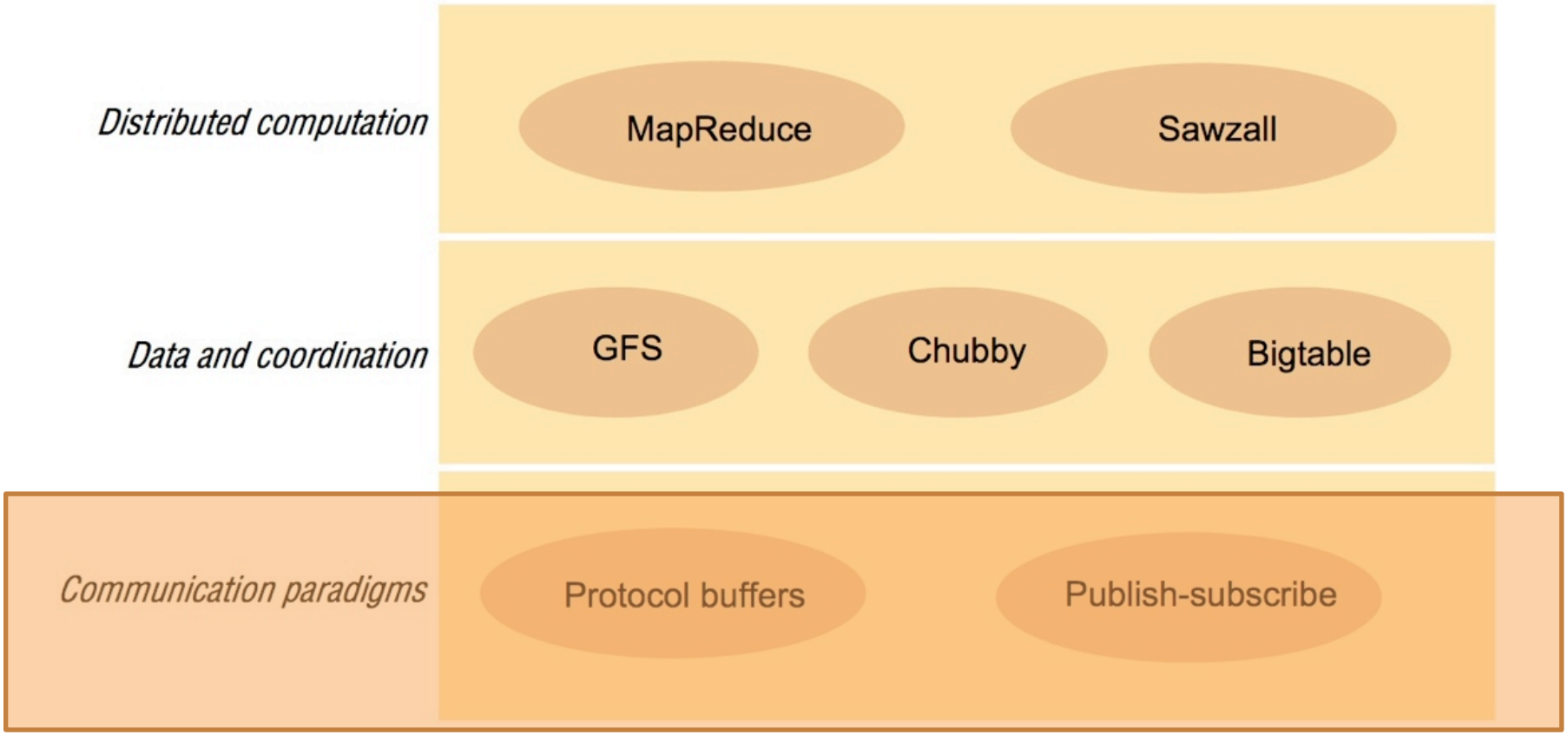
Cluster

Approx 30 racks (around 2400 PCs)
2 high-bandwidth switches (each rack connected to both the switches for redundancy)
Placement and replication generally done at cluster level

Google System Architecture



Google Infrastructure



Google's Communication Paradigms

- Google has developed a set of services that are tailored for various applications running on Google infrastructure

- We will study two important communication paradigms developed by Google
  1. Google Protocol Buffer
  2. Google Publish-Subscribe

Google Protocol Buffer

- Google adopts a minimal and efficient remote invocation service
- Recall that: Remote invocation requires – among all the other services – the following two components
  1. Serialization of data
  2. Agreement on data representation (data-type size and format)
- Protocol Buffer (PB) is a common serialization format for Google

Goal of Protocol Buffer

The goal of Protocol Buffer is to provide a language- and platform-neutral way to specify and serialize data such that:

- Serialization process is efficient, extensible and simple to use
- Serialized data can be stored or transmitted over the network

- In Protocol buffers, Google has designed a language to specify messages

What problem?

For high volume work, no ideal serializer:

- Xml/data-contracts: (relatively) expensive to process; large due to repeated tags
- Soap: as xml, but more
- BinaryFormatter (/NDCS): proprietary closed [non-]standard
- Bespoke: lots of work; lots of potential for error

Protocol Buffer Language

- Message contains uniquely numbered fields
- Field is represented by

<field-type, data-type, field-name, encoding-value, [default value]>

- Available data-types
  - Primitive data-type
    - int, float, bool, string, raw-bytes
  - Enumerated data-type
  - Nested Message
    - Allows structuring data into an hierarchy

```
message Book {
    required string title = 1;
    repeated string author = 2;
    enum Status {
        IN_PRESS = 0;
        PUBLISHED = 1;
        OUT_OF_PRINT = 2;
    }
    message BookStats {
        required int32 sales = 1;
        optional int32 citations = 2;
        optional Status bookstatus = 3 [default = PUBLISHED];
    }
    optional BookStats statistics = 3;
    repeated string keyword = 4;
}
```

Protocol Buffer Language (cont'd)

- Field-types can be:
  - Required fields
  - Optional fields
  - Repeated fields
    - Dynamically sized array
- Encoding-value
  - A unique number (=1,=2,…) represents a tag that a particular field has in the binary encoding of the message

```
message Book {
    required string title = 1;
    repeated string author = 2;
    enum Status {
        IN_PRESS = 0;
        PUBLISHED = 1;
        OUT_OF_PRINT = 2;
    }
    message BookStats {
        required int32 sales = 1;
        optional int32 citations = 2;
        optional Status bookstatus = 3 [default = PUBLISHED];
    }
    optional BookStats statistics = 3;
    repeated string keyword = 4;
}
```

A *.proto* File

- The specification of the message is contained in a *.proto* file
- The *.proto* file is compiled by *protoc* tool
  - The output of the *protoc* is a generated code that allows programmers to manipulate the particular message type
    - For example, assigning, extracting values to/from messages

```
public boolean hasTitle();
public java.lang.String getTitle();
public Builder setTitle(String value);
public Builder clearTitle();
```

- The *Builder* class:
  - Messages are immutable in protocol buffer, Builder class is mutable

An example

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```

An Example

```
Person person;
person.set_name("John Doe");
person.set_id(1234);
person.set_email("jdoe@example.com");
fstream output("myfile", ios::out | ios::binary);
person.SerializeToOstream(&output);
```

```
fstream input("myfile", ios::in | ios::binary);
Person person;
person.ParseFromIstream(&input);
cout << "Name: " << person.name() << endl;
cout << "E-mail: " << person.email() << endl;
```

An Example

```
~/source/protobuf-2.4.1/examples $ ./add_person_cpp haibo-file
haibo-file: File not found.  Creating a new file.
Enter person ID number: 1
Enter name: Haibo
Enter email address (blank for none): haibochen@sjtu.edu.cn
Enter a phone number (or leave blank to finish): 13701830272
Is this a mobile, home, or work phone? mobile
Enter a phone number (or leave blank to finish):
```

```
~/source/protobuf-2.4.1/examples $ ./list_people_cpp haibo-file
Person ID: 1
  Name: Haibo
  E-mail address: haibochen@sjtu.edu.cn
  Mobile phone #: 13701830272
```

Comparison of Protocol Buffer Language

- Advantages of Protocol Buffer (PB)
  - PB is 3-10 times smaller than an XML
  - PB is 10-100 times faster than an XML

- Can we compare PB with XML?
  - PB works only on Google infrastructure, which is relatively closed system and does not address inter-operability
  - XML is richer (it specifies self-describing data and meta-data). PB is not so rich.
    - There are accessory programs that can create a full description.
    - However, they are hardly used

Supporting RPC using Protocol Buffers

PB produces a serialized data that can be used for storage or communications

- Most common use is to use PB for RPCs

Example:

```
service SearchService {
    rpc Search(RequestType) returns (ResponseType)
}
```

- RequestType can correspond to list of keywords
- ResponseType can then correspond to a list of books matching the keywords

protoc compiler takes this specification and produces

- Abstract interface SearchService
- A stub that supports type-safe RPC calls

Extensibility of PB

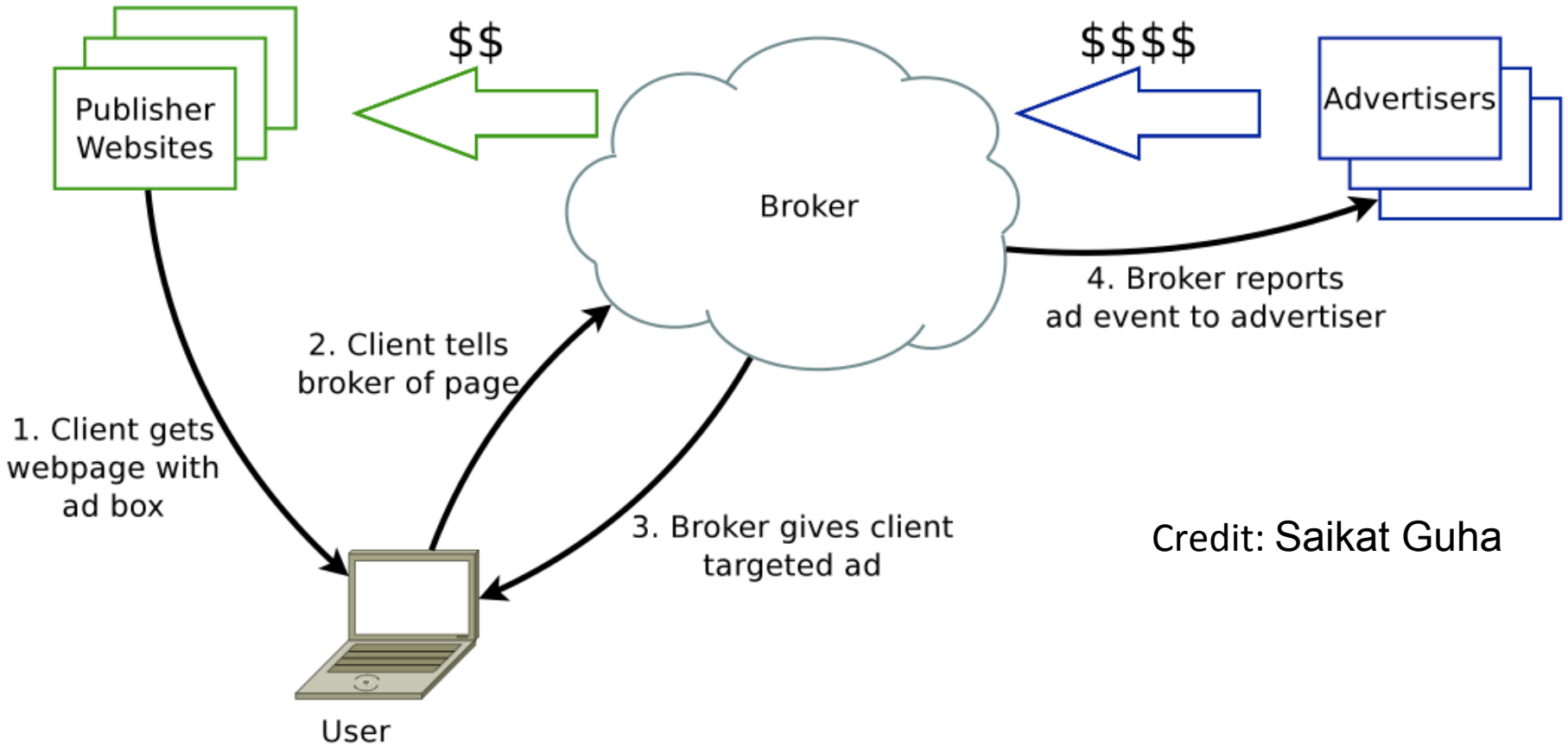
In addition to being language- and platform-neutral, PBs are also agnostic with respect to underlying RPC protocol

PB library provides two abstract interfaces:

- RpcChannel:
  - Provides a common interface to underlying communication
  - e.g., Programmer can specify if HTTP or FTP has to be used for communicating data
- RpcController:
  - Providing common control interface

Google Publish-Subscribe

Ad Infrastructure Today



Credit: Saikat Guha

Question

How to distribute advertisement?

Be distributed in real-time and with reliability guarantees to potentially large number of recipients

Google Publish-Subscribe

Google Publish-Subscribe (PS) is used in applications where *distributed events* need to be distributed in real-time and with reliability guarantees to potentially large number of recipients

PS uses protocol buffers for underlying communication between source, queue and the client

Uses: Google Ads

Unfortunately, Google has not made PS system publicly available

Google Publish-Subscribe

Google adopts a *topic-based* PS system

- A number of channels for event streams with channels corresponding to particular topics

Event contains the following fields:

- Header
- Set of keywords
- Payload: Opaque to the programmer

Subscription request specify

- Channel
- Filter defined over the set of keywords

Channels are used for relatively static and coarse-grained data streams requiring high throughput of events

Google Publish-Subscribe

PS uses a *broker-overlay* in the form of a set of trees, where tree represents a topic

- Root of the tree is the publisher
- Leaf nodes represent subscribers
- Filters are pushed as far back in the tree to minimize the traffic

Google Publish-Subscribe

PS emphasizes strongly on reliable and timely delivery

- Reliability: System maintains redundant trees
  - Two separate tree overlays are maintained for each channel
- Timely delivery: Implements Quality-of-Service management technique to control message flows
- Rate-control is done by imposing limit on per user/per topic event publishing

References

- http://*code.**google**.com/p/protobuf*

Next Class

Programming Distributed Systems:

MapReduce, Dryad(Linq)