Motivating memory-model issues

Tricky and *surprisingly wrong* unsynchronized concurrent code

```
class C {
  private int x = 0;
  private int y = 0;

  void f() {
    x = 1;
    y = 1;
  }
  void g() {
    int a = y;
    int b = x;
    assert(b >= a);
  }
}
```

First understand why it looks like the assertion cannot fail:

- Easy case: call to **g** ends before any call to **f** starts
- Easy case: at least one call to **f** completes before call to **g** starts
- If calls to **f** and **g** *interleave*…

1

What is the problem?

The code has a *data race*

- Two actually
- Recall: data race: unsynchronized read/write or write/write of same location

If code has data races, you cannot reason about it with interleavings!

- That is simply the rules of Java (and C, C++, C#, …)
- (Else would slow down all programs just to "help" programs with data races, and that was deemed a bad engineering trade-off when designing the languages/compilers/hardware)
- So the assertion can fail

2

Why

For performance reasons, the compiler and the hardware often reorder memory operations

Thread 1: **f**

```
x = 1;

y = 1;
```

Thread 2: **g**

```
int a = y;

int b = x;

assert(b >= a);
```

Of course, you cannot just let them reorder anything they want

3

The grand compromise

The compiler/hardware will never perform a memory reordering that affects the result of a single-threaded program

The compiler/hardware will never perform a memory reordering that affects the result of a data-race-free multi-threaded program

So: If no interleaving of your program has a data race, then you can *forget about all this reordering nonsense:* the result will be equivalent to some interleaving

Your job: Avoid data races
Compiler/hardware job: Give illusion of interleaving *if you do your job*

4

Fixing our example

- Naturally, we can use synchronization to avoid data races
  - Then, indeed, the assertion cannot fail

```
class C {
  private int x = 0;
  private int y = 0;
  void f() {
    synchronized(this) { x = 1; }
    synchronized(this) { y = 1; }
  }
  void g() {
    int a, b;
    synchronized(this) { a = y; }
    synchronized(this) { b = x; }
    assert(b >= a);
  }
}
```

5

A second fix

- Java has **`volatile`** fields: accesses do not count as data races
- Implementation: slower than regular fields, faster than locks
- Really for experts: avoid them; use standard libraries instead
- And why do you need code like this anyway?

```
class C {
  private volatile int x = 0;
  private volatile int y = 0;
  void f() {
    x = 1;
    y = 1;
  }
  void g() {
    int a = y;
    int b = x;
    assert(b >= a);
  }
}
```

6

Code that's wrong

- A more realistic example of code that is wrong
  - No *guarantee* Thread 1 will *ever* stop even if "user quits"

```
class C {
  boolean stop = false;
  void f() {
    while(!stop) {
      // draw a monster
    }
  }
  void g() {
    stop = didUserQuit();
  }
}
```

Thread 1: **f()**

Thread 2: **g()**

7

Motivating Deadlock Issues

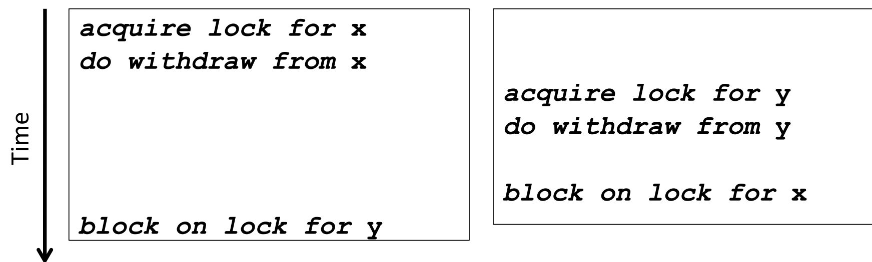
Consider a method to transfer money between bank accounts

```
class BankAccount {
  …
  synchronized void withdraw(int amt) {…}
  synchronized void deposit(int amt) {…}
  synchronized void transferTo(int amt,
                               BankAccount a) {
    this.withdraw(amt);
    a.deposit(amt);
  }
}
```

8

The Deadlock

Suppose **x** and **y** are fields holding accounts



9

Deadlock, in general

A deadlock occurs when there are threads **T1**, …, **Tn** such that:

- For **i**=1,..,n-1, **Ti** is waiting for a resource held by **T(i +1)**
- **Tn** is waiting for a resource held by **T1**

In other words, there is a cycle of waiting

- Can formalize as a graph of dependencies with cycles bad

Deadlock avoidance in programming amounts to techniques to ensure a cycle can never arise

10

Back to our example

Options for deadlock-proof transfer:

1. Make a smaller critical section: **`transferTo`** not synchronized
   - Exposes intermediate state after **`withdraw`** before **`deposit`**
   - May be okay, but exposes wrong total amount in bank
2. Coarsen lock granularity: one lock for all accounts allowing transfers between them
   - Works, but sacrifices concurrent deposits/withdrawals
3. Give every bank-account a unique number and always acquire locks in the same order
   - *Entire program* should obey this order to avoid cycles
   - Code acquiring only one lock can ignore the order

11

Ordering locks

```
class BankAccount {
  …
  private int acctNumber; // must be unique
  void transferTo(int amt, BankAccount a) {
    if(this.acctNumber < a.acctNumber)
       synchronized(this) {
       synchronized(a) {
          this.withdraw(amt);
          a.deposit(amt);
       }}
    else
       synchronized(a) {
       synchronized(this) {
          this.withdraw(amt);
          a.deposit(amt);
       }}
  }
}
```

12

Another example

From the Java standard library

```
class StringBuffer {
  private int count;
  private char[] value;
  …
  synchronized append(StringBuffer sb) {
    int len = sb.length();
    if(this.count + len > this.value.length)
      this.expand(…);
    sb.getChars(0,len,this.value,this.count);
  }
  synchronized getChars(int x, int, y,
                        char[] a, int z) {
    "copy this.value[x..y] into a starting at z"
  }
}
```

13

Reading vs. writing

Recall:

- Multiple concurrent reads of same memory: *Not* a problem
- Multiple concurrent writes of same memory: Problem
- Multiple concurrent read & write of same memory: Problem

So far:

- If concurrent write/write or read/write might occur, use synchronization to ensure one-thread-at-a-time

But this is unnecessarily conservative:

- Could still allow multiple simultaneous readers!

14

Example

Consider a hashtable with one coarse-grained lock
- So only one thread can perform operations at a time

But suppose:
- There are many simultaneous `lookup` operations
- **`insert`** operations are very rare

Note: Important that **`lookup`** does not actually mutate shared memory, like a move-to-front list operation would

15

Readers/writer locks

A new synchronization ADT: The readers/writer lock

- A lock's states fall into three categories:
  - "not held"
  - "held for writing" by one thread
  - "held for reading" by *one or more* threads

$0 \leq$ writers ≤ 1
$0 \leq$ readers
writers*readers==0

- **`new:`** make a new lock, initially "not held"
- **`acquire_write:`** block if currently "held for reading" or "held for writing", else make "held for writing"
- **`release_write:`** make "not held"
- **`acquire_read:`** block if currently "held for writing", else make/keep "held for reading" and increment *readers count*
- **`release_read:`** decrement readers count, if 0, make "not held"

16

Pseudocode example (not Java)

```
class Hashtable<K,V> {
  …
  // coarse-grained, one lock for table
  RWLock lk = new RWLock();
  V lookup(K key) {
    int bucket = hasher(key);
    lk.acquire_read();
    … read array[bucket] …
    lk.release_read();
  }
  void insert(K key, V val) {
    int bucket = hasher(key);
    lk.acquire_write();
    … write array[bucket] …
    lk.release_write();
  }
}
```


Readers/writer lock details

- A readers/writer lock implementation ("not our problem") usually gives *priority* to writers:
  - Once a writer blocks, no readers *arriving later* will get the lock before the writer
  - Otherwise an **`insert`** could *starve*
- Re-entrant?
  - Mostly an orthogonal issue
  - But some libraries support *upgrading* from reader to writer
- Why not use readers/writer locks with more fine-grained locking, like on each bucket?
  - Not wrong, but likely not worth it due to low contention

In Java

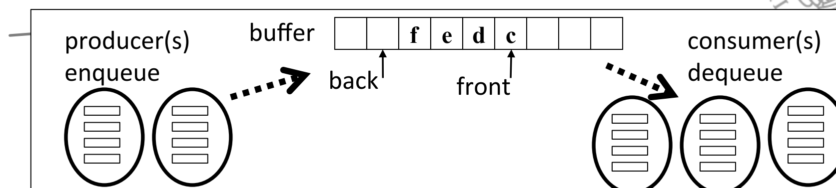
Java's **synchronized** statement does not support readers/writer

Instead, library
java.util.concurrent.locks.ReentrantReadWriteLock

- Different interface: methods **readLock** and **writeLock** return objects that themselves have **lock** and **unlock** methods
- Does *not* have writer priority or reader-to-writer upgrading
  - Always read the documentation

19

Motivating Condition Variables



To motivate condition variables, consider the canonical example of a bounded buffer for sharing work among threads

Bounded buffer: A queue with a fixed size

For sharing work – think an assembly line:
- Producer thread(s) do some work and enqueue result objects
- Consumer thread(s) dequeue objects and do next stage
- Must synchronize access to the queue

20

Code, attempt 1

```
class Buffer<E> {
  E[] array = (E[])new Object[SIZE];
  … // front, back fields, isEmpty, isFull methods
  synchronized void enqueue(E elt) {
    if(isFull())
      ???
    else
      … add to array and adjust back …
  }
  synchronized E dequeue()
    if(isEmpty())
      ???
    else
      … take from array and adjust front …
  }
}
```

21

Waiting

- **enqueue** to a full buffer should *not* raise an exception
  - Wait until there is room
- **dequeue** from an empty buffer should *not* raise an exception
  - Wait until there is data

Bad approach is to *spin* (wasted work and keep grabbing lock)

```
void enqueue(E elt) {
  while(true) {
    synchronized(this) {
      if(isFull()) continue;
      … add to array and adjust back …
      return;
}}}
// dequeue similar
```

22

What we want

- Better would be for a thread to *wait* until it can proceed
  - Be *notified* when it should try again
  - In the meantime, let other threads run
- Like locks, not something you can implement on your own
  - Language or library gives it to you, typically implemented with operating-system support
- An ADT that supports this: condition variable
  - Informs waiter(s) when the *condition* that causes it/them to wait has *varied*
- Terminology not completely standard; will mostly stick with Java

23

Java approach: **not** quite right

```
class Buffer<E> {
  …
  synchronized void enqueue(E elt) {
    if(isFull())
      this.wait(); // releases lock and waits
    add to array and adjust back
    if(buffer was empty)
      this.notify(); // wake somebody up
  }
  synchronized E dequeue() {
    if(isEmpty())
      this.wait(); // releases lock and waits
    take from array and adjust front
    if(buffer was full)
      this.notify(); // wake somebody up
  }
}
```

24

Key ideas

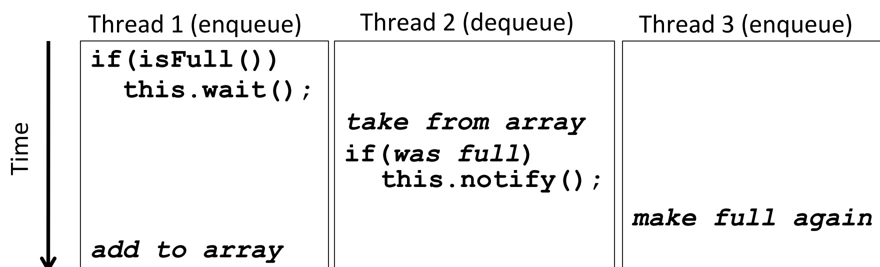
- Java weirdness: every object "is" a condition variable (and a lock)
  - other languages/libraries often make them separate
- **`wait:`**
  - "register" running thread as interested in being woken up
  - then atomically: release the lock and block
  - when execution resumes, *thread again holds the lock*
- **`notify:`**
  - pick one waiting thread and wake it up
  - no guarantee woken up thread runs next, just that it is no longer blocked on the *condition* – now waiting for the *lock*
  - if no thread is waiting, then do nothing

25

Bug #1

```
synchronized void enqueue(E elt){
  if(isFull())
    this.wait();
  add to array and adjust back
  …
}
```

Between the time a thread is notified and it re-acquires the lock, the condition can become false again!



26

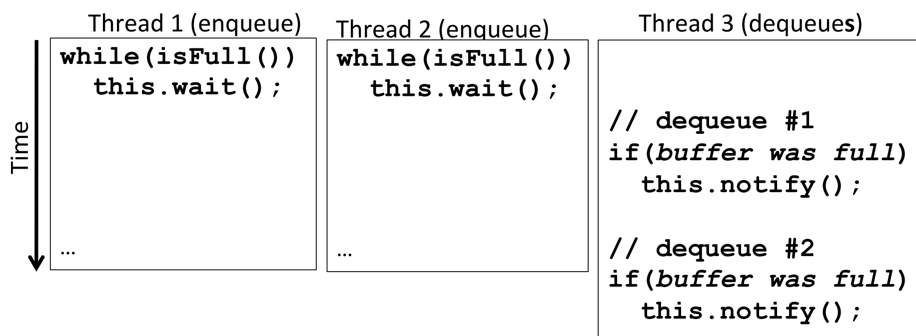
Bug fix #1

```
synchronized void enqueue(E elt) {
  while(isFull())
    this.wait();
  …
}
synchronized E dequeue() {
  while(isEmpty())
    this.wait();
  …
}
```

Guideline: *Always* re-check the condition after re-gaining the lock

Bug #2

- If multiple threads are waiting, we wake up only one
  - Sure only one can do work *now*, but can't forget the others!



Bug fix #2

```
synchronized void enqueue(E elt) {
  …
  if(buffer was empty)
    this.notifyAll(); // wake everybody up
}
synchronized E dequeue() {
  …
  if(buffer was full)
    this.notifyAll(); // wake everybody up
}
```

`notifyAll` wakes up all current waiters on the condition variable

Guideline: If in any doubt, use **`notifyAll`**
- Wasteful waking is better than never waking up

- So why does **`notify`** exist?
  - Well, it is faster when correct…

Last condition-variable comments

- `notify`/**`notifyAll`** often called **`signal/broadcast`**, also called **`pulse/pulseAll`**
- Condition variables are subtle and harder to use than locks
- But when you need them, you need them
  - Spinning and other work-arounds do not work well
- Fortunately, like most things in a data-structures course, the common use-cases are provided in libraries written by experts
  - Example: **`java.util.concurrent.ArrayBlockingQueue<E>`**
  - All uses of condition variables hidden in the library; client just calls **`put`** and **`take`**

Concurrency summary

- Access to shared resources introduces new kinds of bugs
  - Data races
  - Critical sections too small
  - Critical sections use wrong locks
  - Deadlocks
- Requires synchronization
  - Locks for mutual exclusion (common, various flavors)
  - Condition variables for signaling others (less common)
- Guidelines for correct use help avoid common pitfalls
- Not clear shared-memory is worth the pain
  - But other models (e.g., message passing) not a panacea

31

DISTRIBUTED PROCESSING

32

Distributed Processing

- distributed processing: the execution of concurrent processes by running them on separate processors which communicate by message passing.
- Our view: language-based approach

33

Assumptions

- processors share only a communication network,
- the processes don't share a common address space, so they can't communicate via shared variables instead they communicate by sending and receiving messages

34

Message Passing

- Processes communicate by sending and receiving messages using special message passing primitives which include synchronisation:
  - **send (destination) message**: sends message to another process
  - **receive (source) message**: indicates that a process is ready to receive a message message from another process source

35

- **asynchronous communication**: the sending process continues without waiting for the message to be received, e.g., Unix sockets, java.net
- **synchronous communication**: the sending process is delayed until the corresponding receive is executed, e.g., CSP, occam
- **remote invocation**: the sending process is delayed until a reply is received, e.g., RPC (java.rmi), Extended Rendezvous

36

Remote invocation

- With *remote invocation* a process executes a synchronous send and waits until the reply is received:
- combines aspects of monitors and synchronous message passing:
  - as with monitors interaction is via public procedures
  - as with synchronous send, calling a procedure delays the caller
  - provides two way communication from the caller to the process servicing the call and back

37

Two forms

- **Remote Procedure Call** creates a new process to handle each call
- **Extended Rendezvous** services a request using an existing process

38

Java RMI

- The package java.rmi implements Java's version of RPC: remote invocation is based on the model of a procedure call
- in Java, non-static methods must be invoked on an object
- Java therefore requires both remote methods (procedures) and remote objects on which the remote methods can be invoked.

39

Remote Objects

- A Java remote object is one whose methods can be invoked from another JVM, potentially on a different host:
- a remote object is described by one or more remote interfaces which extend java.rmi.Remote
- methods declared in a Remote interface must throwRemoteExceptions
- remote method invocation (RMI) is the action of invoking a method of a remote interface on a remote object

40

Structure of RMI Apps

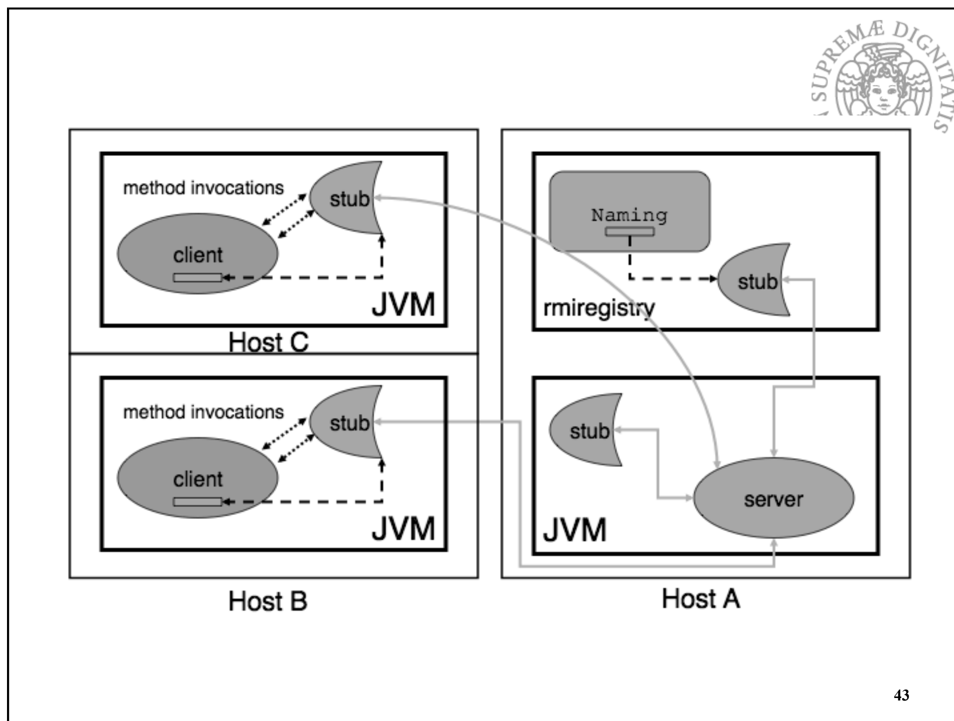
- a server creates some remote objects, makes references to them accessible and waits for clients to invoke (remote) methods on the remote objects
- a client gets a remote reference to a remote object in the server, either from the RMI registry or as a return value to a remote method, and invokes (remote) methods on it
- a component of a distributed Java application can act as both a client and server

41

RMI Registry

- The system provides a particular remote object, the RMI registry for finding references to remote objects:
- once a remote object is registered with the RMI registry on the local host, clients on any host can look up the remote object by name, obtain a reference to it (stub), and then invoke its methods
- the registry is typically used only to locate the first remote object that a client needs to use from a particular server
- the registry listens on a known port, usually 1099 on the same host as the server.

42



43

STUB

- A *stub* acts as a proxy for a remote object and is responsible for carrying out method calls on the remote object.
- Invoking a stub method:
  - initiates a connection with the remote JVM containing the remote object;
  - writes and transmits the method parameters to the remote JVM;
  - waits for the results of the method invocation; and
  - reads the result (return value or exception) and returns it to the caller

44

Parameter passing

- An argument to or return value form a remote object can be any Java object that is *serializable*:
  - non-remote method arguments and results are passed by *copying*– changes made to the object are not visible to other clients.
  - remote objects are passed by *reference* (i.e., a copy of the stub is passed or returned)–changes made by one client to the state of the remote object are visible to all clients.

45

```
class RWDictionary {
    private final Map<String, Data> m =
      new TreeMap<String, Data>();

    // locks
    private final ReentrantReadWriteLock rwl =
      new ReentrantReadWriteLock();
    private final Lock r = rwl.readLock();
    private final Lock w = rwl.writeLock();

    // methods follow ...
```

46

```
// Reader method (does not update the map)
public Data get(String key) {
     r.lock();
     try {
          return m.get(key);
     }
     finally {
          r.unlock();
     }
 }
```

47

```
    // Writer method (changes the map)
    public Data put(String key, Data value) {
         w.lock();
         try {
              return m.put(key, value);
         }
         finally {
              w.unlock();
         }
    }
}
```

48

```
import java.rmi.*;
import java.rmi.server.*;

interface RWDictionaryServer extends Remote {

    Data get(String key) throws RemoteException;

    Data put(String key, Data value) throws
      RemoteException;
}
```

49

```
class RWDictionaryServerImpl
  extends UnicastRemoteObject implements RWDictionaryServer {

    private final Map<String, Data> m =
      new TreeMap<String, Data>();

    private final ReentrantReadWriteLock rwl =
      new ReentrantReadWriteLock();
    private final Lock r = rwl.readLock();
    private final Lock w = rwl.writeLock();

    public RWDictionaryServerImpl() throws RemoteException { }

    // continued ...
```

50

```
// Reader method (does not update the map)
public Data get(String key)
  throws RemoteException {
     r.lock();
     try {
         return m.get(key);
     }
     finally {
         r.unlock();
     }
 }
```

51

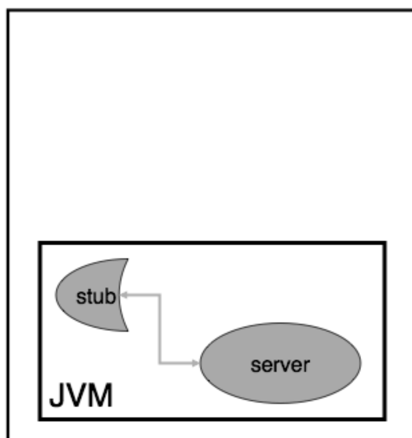
```
    // Writer method (changes the map)
    public Data put(String key, Data value)
     throws RemoteException {
        w.lock();
        try {
            return m.put(key, value);
        }
        finally {
            w.unlock();
        }
    }
}
```

52

```
    public static void main(String[] args) {
        try {
            RWDictionaryServer server =
              new RWDictionaryServerImpl();
            Naming.bind("//host:port/rwDictionary", server);
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

53

```
    public static void main(String[] args) {
        try {
            RWDictionaryServer server =
              new RWDictionaryServerImpl();
            Naming.bind("//host:port/rwDictionary", server);
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

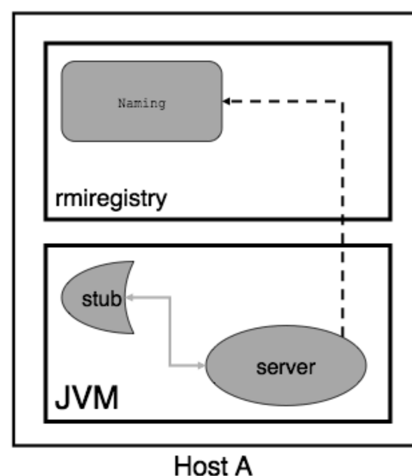


54

- Before a caller can invoke a method on a remote object, it must obtain a remote reference to it:
  - the Naming interface is used for registering and looking up remote objects in the *registry*
  - once a remote object is registered with the RMI registry on the local host, clients on any host can look up the remote object by name, obtain its reference and then invoke its methods.

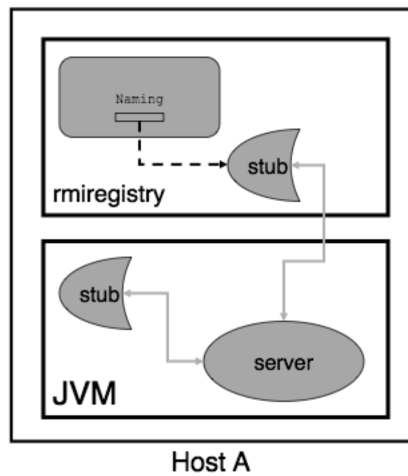
55

Naming.bind



56

Downloading the stub

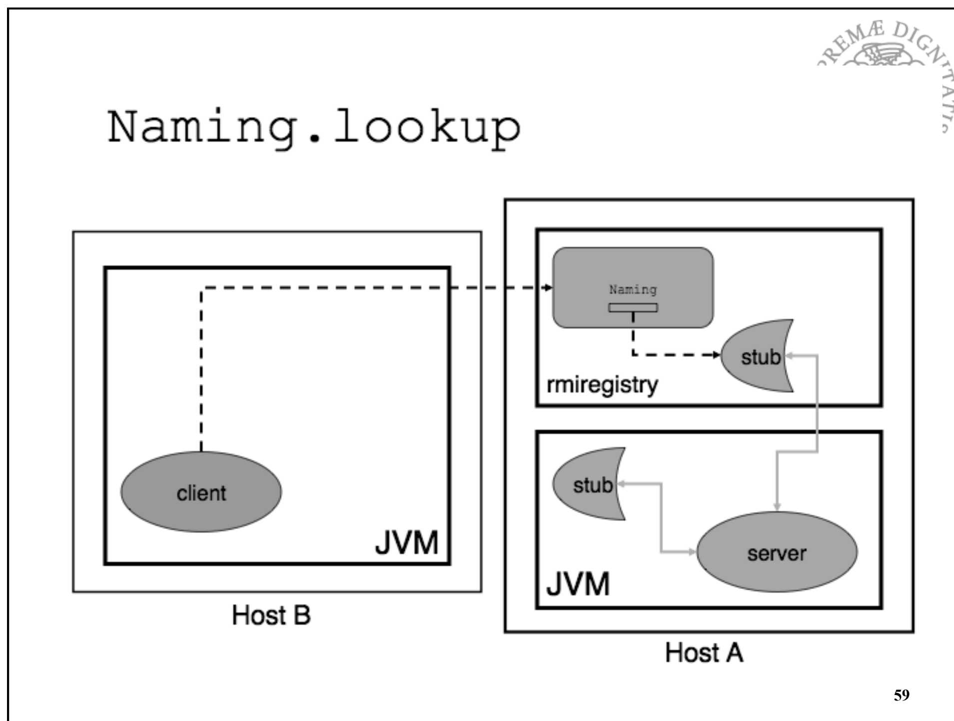


57

WriterClient

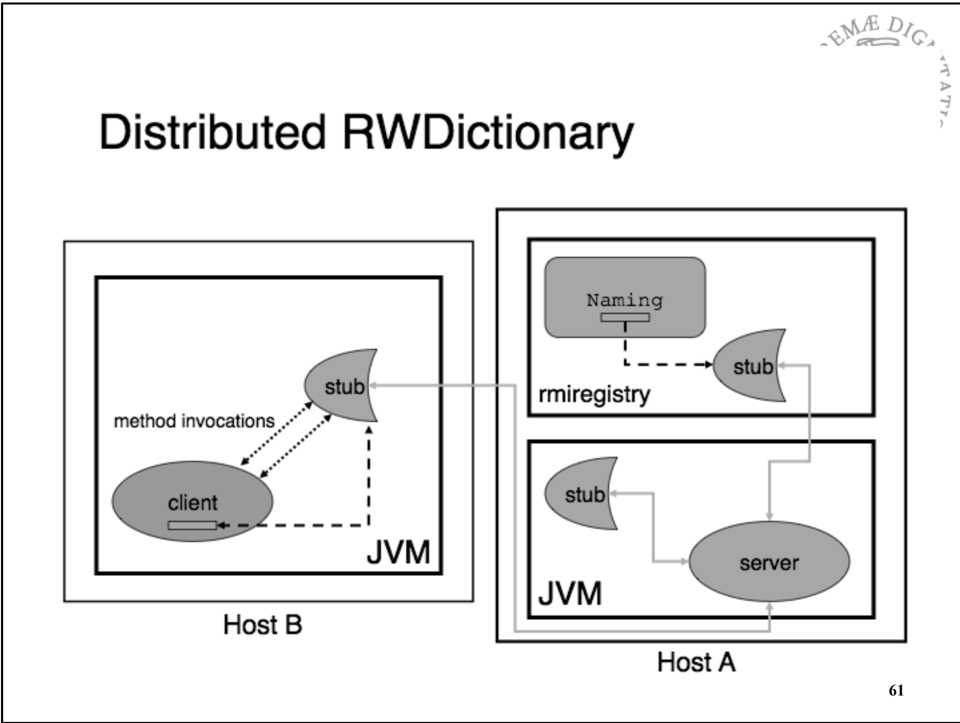
```
class WriterClient {
    public static void main(String[] args) {
        try {
            System.setSecurityManager(new RMISecurityManager());
            String name = "//host:port/rwDictionary";
            RWDictionaryServer d =
              (RWDictionaryServer) Naming.lookup(name);

            // Make a key,value pair and put it in the dictionary
            d.put(key, value);
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

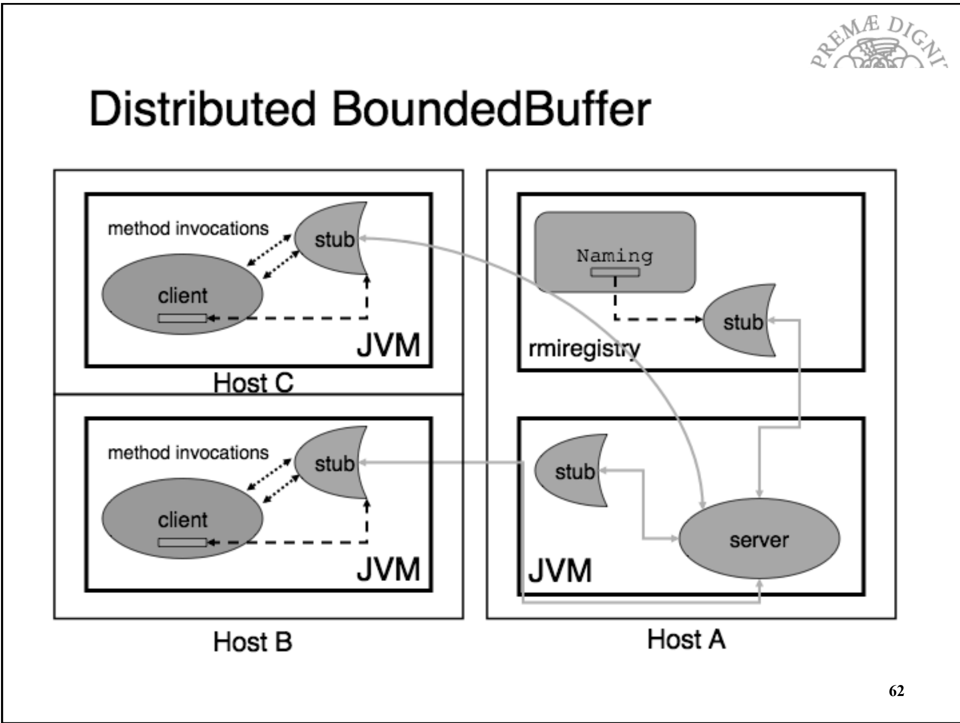
58



- the RWDictionaryServerImpl_Stub is downloaded to the client's JVM from the registry's web server
- the stub knows the anonymous port on which the RWDictionaryServer_Impl is listening for method calls
- the WriterClient can then invoke methods on the stub, e.g., put(String key, Data value)

60


Distributed RWDictionary
Naming
stub
rmiregistry
stub
method invocations
client
JVM
Host B
stub
server
JVM
Host A
61


Distributed BoundedBuffer
method invocations
stub
client
JVM
Host C
method invocations
stub
client
JVM
Host B
Naming
stub
rmiregistry
stub
server
JVM
Host A
62

RMI and Threads

"A method dispatched by the RMI runtime to a remote object may or may not execute in a separate thread. The RMI runtime makes no guarantees with respect to mapping remote method invocations to threads. Since remote method invocation on the same remote method may execute concurrently, **a remote object implementation needs to make sure its implementation is thread-safe**."

— Java RMI Specification, section 3.2

63