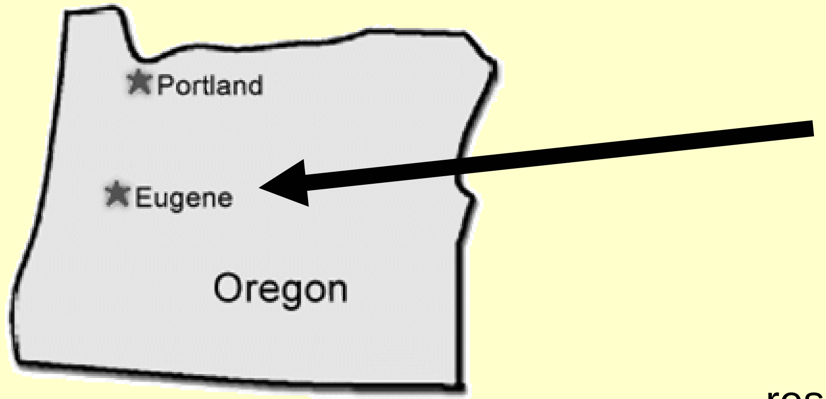
Language Tools for Distributed Computing and Program Generation



Yannis Smaragdakis
University of Oregon

(with a cast of many: credits at the end)

research supported by NSF grants CCR-0220248 and CCR-0238289, LogicBlox Inc.

My Research

- ***The systems and languages end of SE***
  - **language tools for distributed computing**
    - NRMI, J-Orchestra, GOTECH
  - **automatic testing**
    - JCrasher, Check-n-Crash (CnC), DSD-Crasher
  - **program generators and domain-specific languages**
    - MJ, cJ, Meta-AspectJ (MAJ), SafeGen, JTS, DiSTiL
  - **multiparadigm programming**
    - FC++, LC++
  - **software components**
    - mixin layers, layered libraries
  - **memory management**
    - EELRU, compressed VM, trace reduction, adaptive replacement

These Lectures

- NRMI: middleware offering a natural programming model for distributed computing
  - solves a long standing, well-known open problem!
- J-Orchestra: execute unsuspecting programs over a network, using program rewriting
  - led to key enhancements of a major open-source software project (JBoss)
- Morphing: a high-level language facility for safe program transformation
  - "bringing discipline to meta-programming"

This Talk

- NRMI: middleware offering a natural programming model for distributed computing
  - solves a long standing, well-known open problem!
- J-Orchestra: execute unsuspecting programs over a network, using program rewriting
  - led to key enhancements of a major open-source software project (JBoss)
- Morphing: a high-level language facility for safe program transformation
  - “bringing discipline to meta-programming”

Language Tools for Distributed Computing

- What does “language tools” mean?
  - middleware libraries, compiler-level tools, program generators, domain-specific languages
- What is a distributed system?
  - “A collection of independent computers that appears to users as a single, coherent system”

Why Language Tools for Distributed Computing?

- Why Distributed Computing?
  - networks changed the way computers are used
  - programming distributed systems is hard!
    - partial failure, different semantics (distinct memory spaces), high latency, natural multi-threading
  - are there simple programming models to make our life easier?
- *"The future is distributed computation, but the language community has done very little to address that possibility."*

Rob Pike—"Systems Software Research is Irrelevant", 2000

A Bit of Philosophy
(of Distributed Systems, of course)

"A Note on Distributed Computing"
(Waldo, Wyant, Wollrath, Kendall)

Highly influential 1994 manifesto for distributed systems programming

Main Thesis of “Note”

- Main thesis of the paper: *distributed* computing is very different from *local* computing
- We shouldn’t be trying to make one resemble the other
- We cannot hide the specifics of whether an object is distributed or local (“paper over” the network)
- Distributing objects *cannot* be an afterthought
  - there are often dependencies in an object’s interface that determine whether it can be remote or not
- The “vision of unified objects” contains fallacies

Vision of Unified Objects

- What is it?
  - Design and implement your application, without consideration of whether objects are local or remote
  - Then, choose object locations and interfaces for performance
  - Finally, expand objects to deal with partial failures (e.g., network outages) by adding replication, transactions, etc.

“Note” argument

- The premise of “unified object” is wrong:
  - the design of an application is dependent on whether it is local or remote
  - the implementation is dependent on whether it is local or remote
  - the interfaces to objects are dependent on whether objects are local or remote

Differences between Local and Distributed Computing

- Latency, memory access, partial failure, and concurrency
  - Latency: remote operations take much longer to complete than local ones
  - Memory access: cannot access remote memory directly (e.g., with pointers)
  - Partial failure and concurrency: remote operations may fail, or parts of them may fail. Also, distributed objects can be accessed concurrently and need to synchronize

How Do Differences Affect Programming?

- Latency:
  - if ignored leads to performance problems
  - important, but critical?
    - can be alleviated with judicious object placement
- Memory access:
  - “it would be too restrictive to prevent programmers from manipulating memory through pointers”
  - Things have changed a lot. Java papers over memory *and* makes everything be an object. Hence, it’s all a matter of defining the right abstractions

The Big One

- Partial failure and concurrency:
  - more serious problems, as operations fail often, and sometimes parts of them succeed and cause later trouble
  - this is an important factor!

Dealing with Partial Failure

- We can either
  - treat all objects as local objects

  or
  - treat all objects as distributed objects
- Problems:
  - The former cannot handle failure well
  - The latter is a non-solution: instead of making distributed computing as simple as local, we make local computing as hard as distributed
- The same holds for concurrency!

Some Great Examples

- Imagine a “queue” data structure object
  - interface:
    - enqueue(object), dequeue(object), etc.
  - the queue is held remotely
- Problems:
  - on timeout, should I re-insert?
    - what if insertion fails completely?
    - what if insertion succeeded but confirmation was not received?
  - how do I avoid duplication?
    - need request identifiers, but the queue interface does not support them!

Partial Failure and Interfaces

- In short, recovery from partial failure cannot be an afterthought. Implementation choices are apparent in the client interface. No “ideal” interface is suitable for all implementations.
- Same for performance (example of set and testing object equality)

Case Study

- Consider NFS (network file system)
- *soft mounts* signal client programs (e.g., your regular, everyday executable) when a file system operation fails
  - result: applications crash
- *hard mounts* just block until operation terminates
  - result: machines freeze too easily, complex interdependencies arise

NFS Case Study

- The “Note” argues that the interface (read, write, etc. syscalls) upon which NFS is built does not lend itself to distributed implementations
  - “the reliability of NFS cannot be changed without a change to that interface”

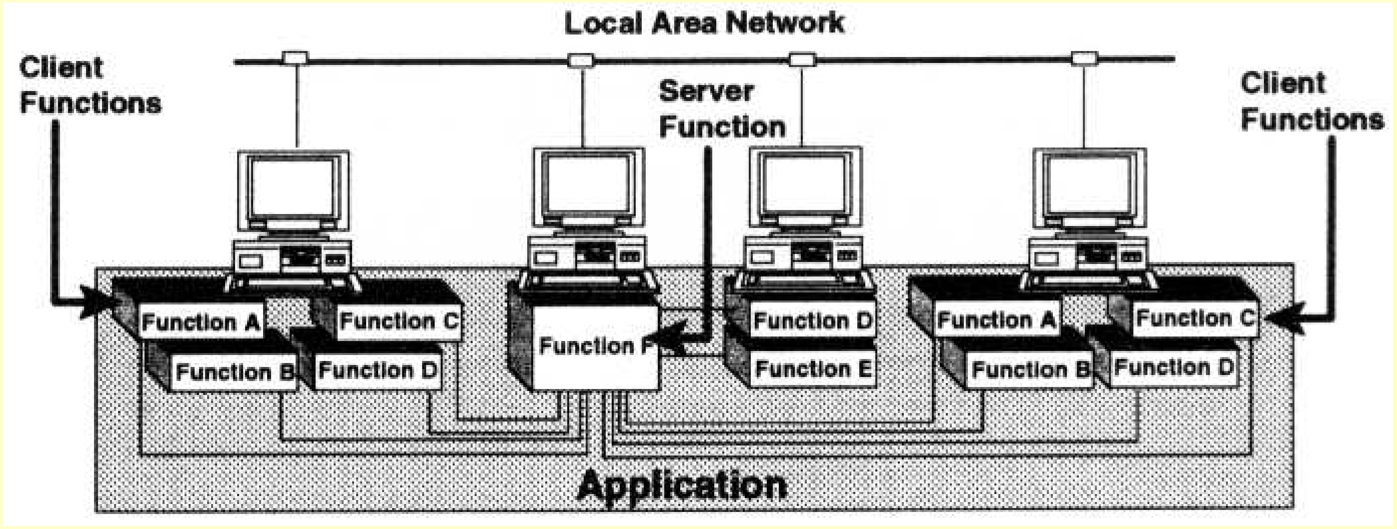
And Despite All That...

- NFS seems to be a good example for both the paper’s argument and the opposite:
  - the read, write, etc. syscall interface is great for applications, because it masks the local/remote aspects
  - NFS is successful *because* of the interface, not in spite of it!
  - at a lower level, NFS should indeed be implemented in a distributed fashion (e.g., with transactions and replication)
    - NFS could be improved, without changing the interface (contrary to the paper’s assertion)

How Can we Hide Distribution

while leaving control with the programmer?

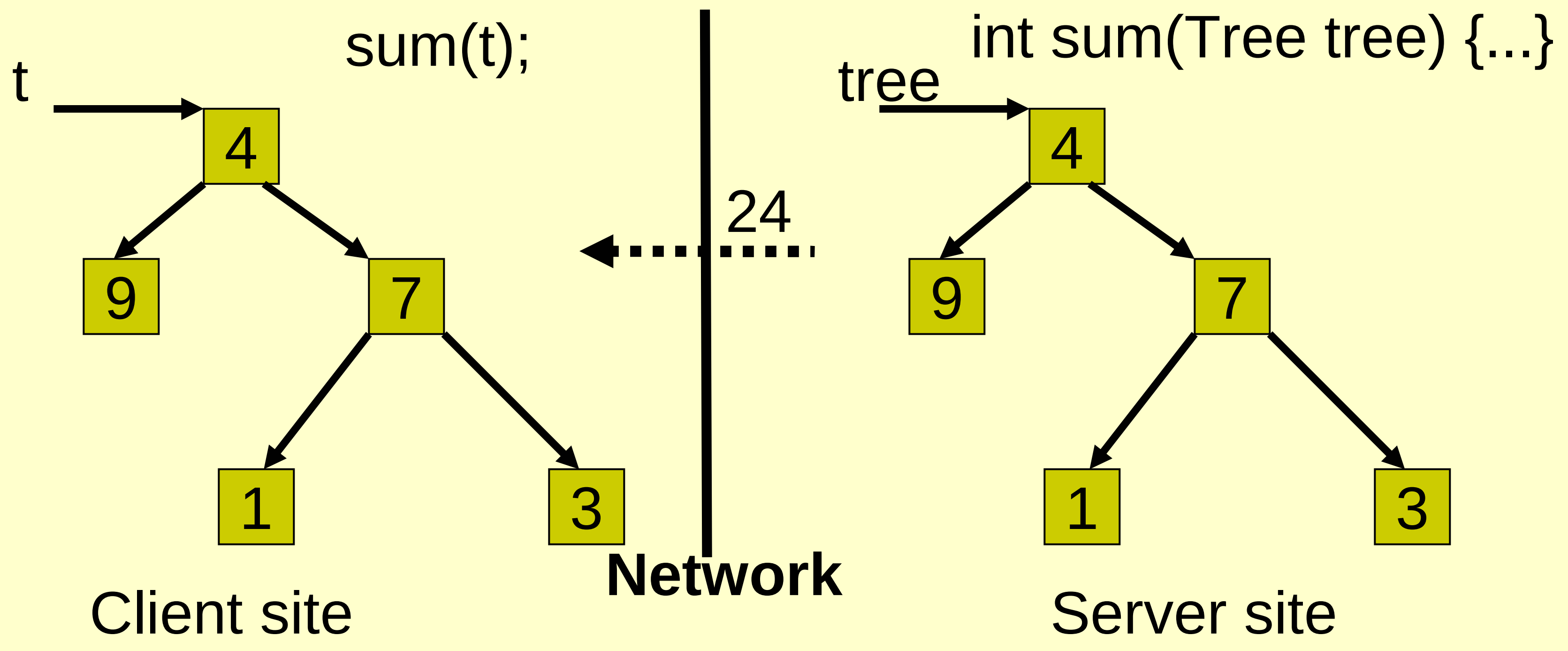
Programming Distributed Systems



- A very common model is RPC middleware:
  - hide network communication behind a procedure call (“remote procedure call”)
  - execute call on server, but make it look to client like a local call
    - only, not quite: need to be aware of different memory space
- Our problem: make RPC calls more like local calls!

Common RPC Programming Model (*call semantics*): Call-by-copy

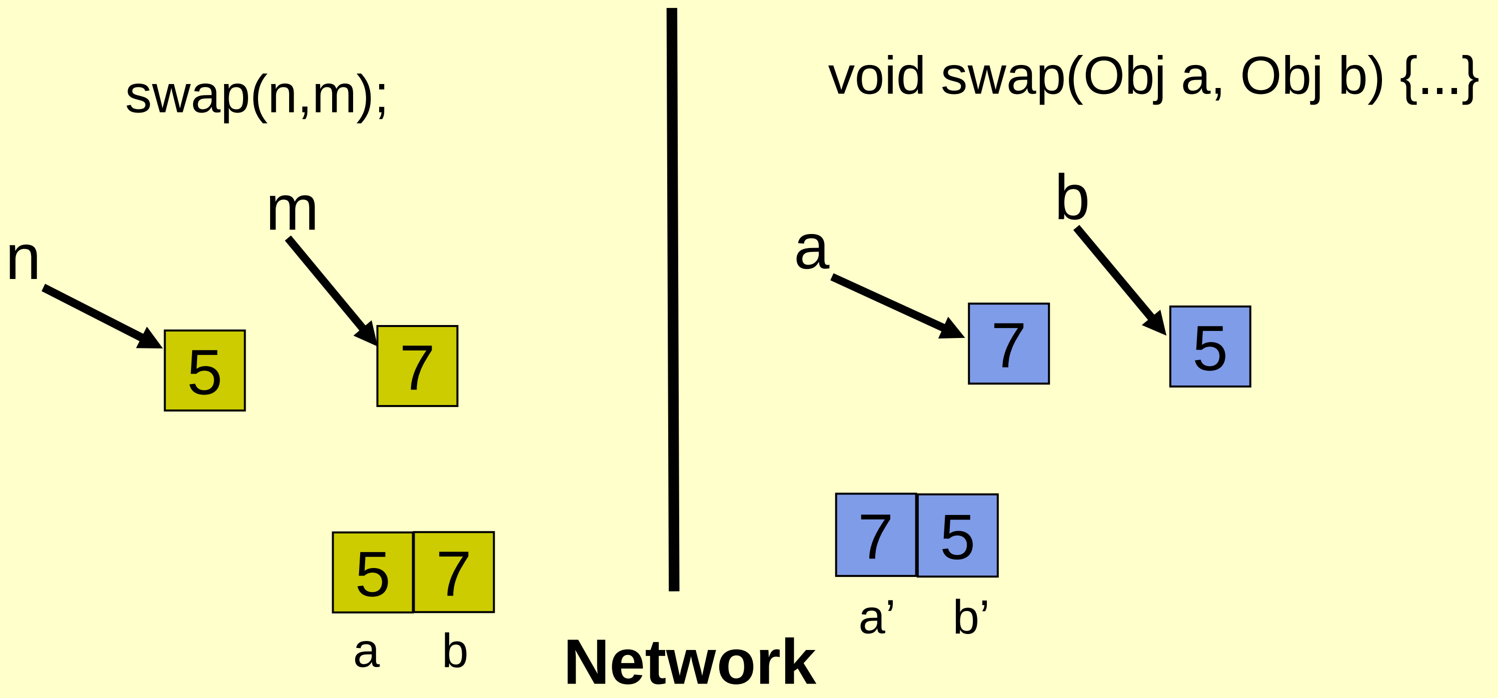
- To call a remote procedure, copy argument-reachable data to server site, return value back
    - data packaged and sent over net (“pickling”, “serialization”)



Other Calling Semantics: Call-by-Copy-Restore

- Call-by-copy (*call-by-value*) works fine when the remote procedure does not need to modify arguments
  - otherwise, changes not visible to caller, unlike local calls
  - in general, not easy to change shared state with non-shared address spaces
- *Call-by-copy-restore* is a common idea in distributed systems (and in some languages, as *call-by-value-result*):
  - copy arguments to remote procedure, copy results of execution back, restore them in original variables
  - resembles call-by-reference on a single address space

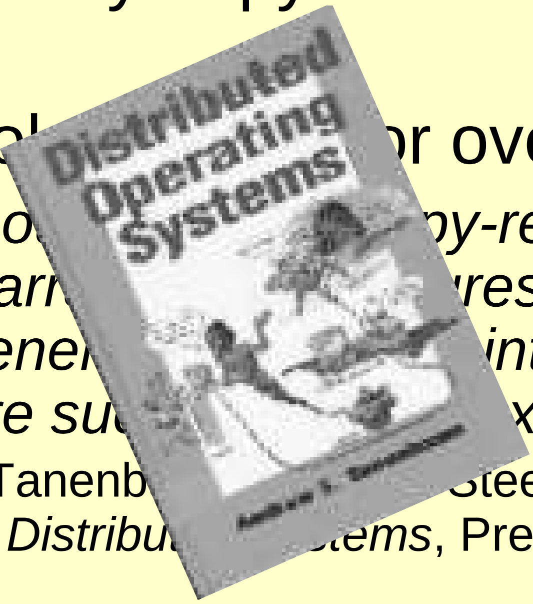
Copy-Restore Example

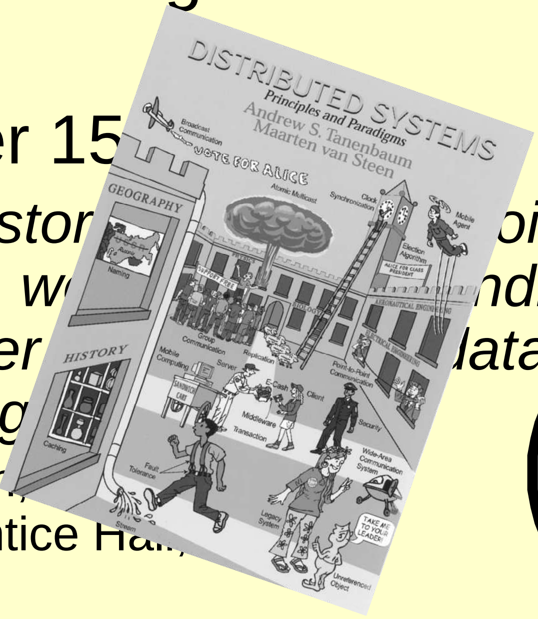


A Long Standing Challenge

- Works ok for single variables, but not complex data!
- The distributed systems community has long tried to define call-by-copy-restore as a general model, for all data
- A textboo or over 15
  - *"… Altho py-restor pointers to simple arr ures, w ndle the most gene inter data structure su x g*

    Tanenb Steen,
    Distribu stems, Prentice Hal





  - The DCE RPC design tried to solve it but did not

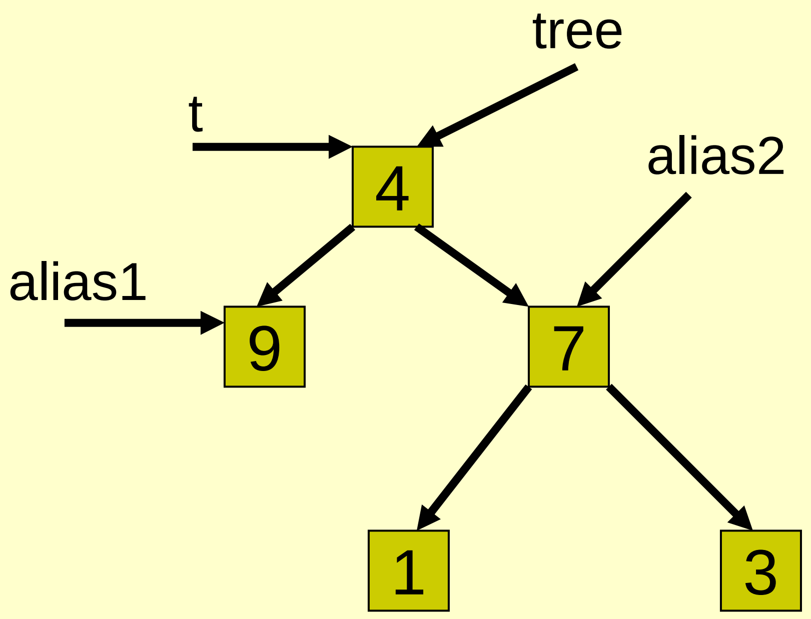
Our Contribution: NRMI

- The NRMI ("Natural RMI") middleware facility solves the general problem *efficiently*
  - a drop-in replacement of Java RMI, also supporting full call-by-copy-restore semantics
  - *invariant: all changes from the server are visible to client when RPC returns*
    - no matter what data are used and how they are linked
    - this is the hallmark property of copy-restore
- The difficulty:
  - having pointers means having *aliasing*: multiple ways to reach the same object—need to correctly update all

Solution Idea (by example)

- Consider what changes a procedure can make

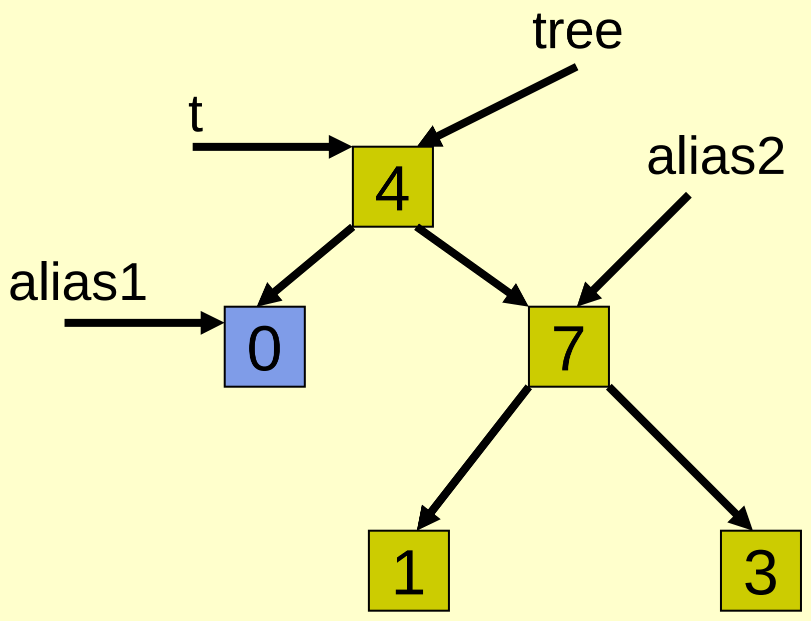
```
foo(t); ...

void foo (Tree tree) {
 tree.left.data = 0;
 tree.right.data = 9;
 tree.right.right.data = 8;
 tree.left = null;
 Tree temp =
   new Tree(2, tree.right.right, null);
 tree.right.right = null;
 tree.right = temp;
}
```



Solution Idea (by example)

- Consider what changes a procedure can make

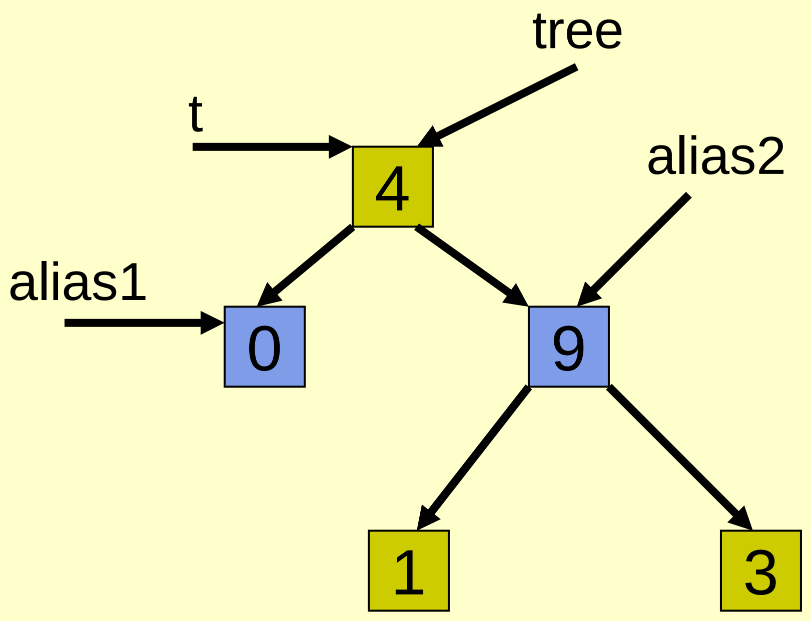
```
foo(t); ...

void foo (Tree tree) {
 tree.left.data = 0;
 tree.right.data = 9;
 tree.right.right.data = 8;
 tree.left = null;
 Tree temp =
    new Tree(2, tree.right.right, null);
 tree.right.right = null;
 tree.right = temp;
}
```



Solution Idea (by example)

- Consider what changes a procedure can make

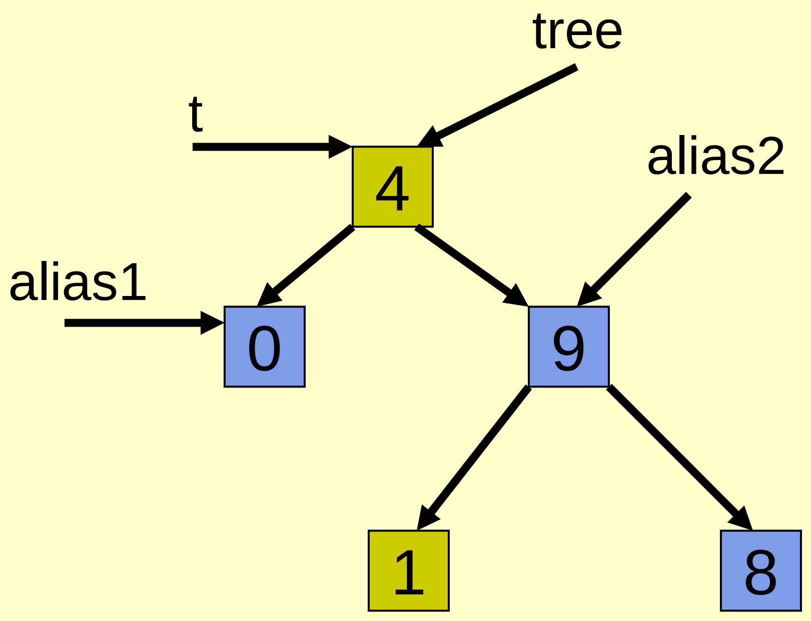
```
foo(t); ...

void foo (Tree tree) {
 tree.left.data = 0;
 tree.right.data = 9;
 tree.right.right.data = 8;
 tree.left = null;
 Tree temp =
    new Tree(2, tree.right.right, null);
 tree.right.right = null;
 tree.right = temp;
}
```



Solution Idea (by example)

- Consider what changes a procedure can make

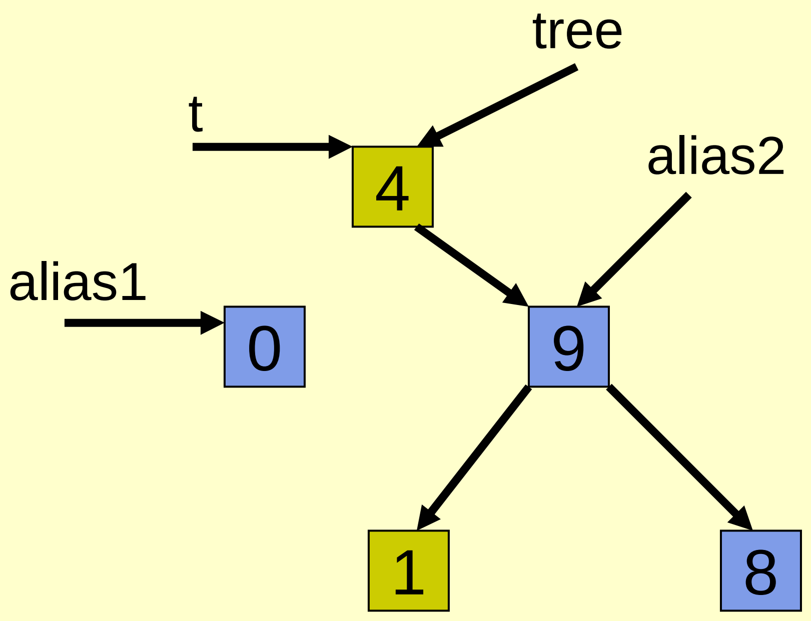
```
foo(t); ...

void foo (Tree tree) {
 tree.left.data = 0;
 tree.right.data = 9;
 tree.right.right.data = 8;
 tree.left = null;
 Tree temp =
   new Tree(2, tree.right.right, null);
 tree.right.right = null;
 tree.right = temp;
}
```



Solution Idea (by example)

- Consider what changes a procedure can make

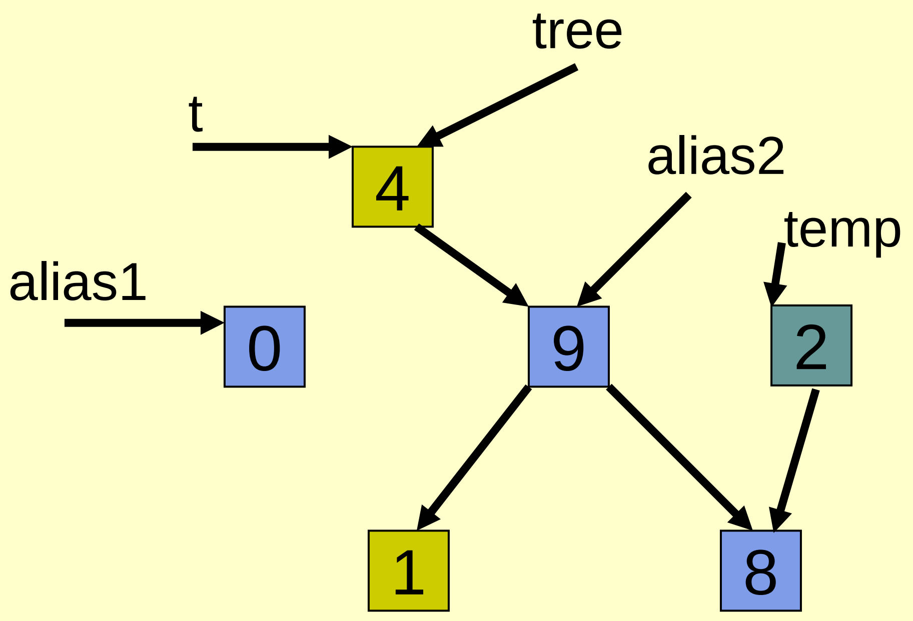
```
foo(t); ...

void foo (Tree tree) {
 tree.left.data = 0;
 tree.right.data = 9;
 tree.right.right.data = 8;
 tree.left = null;
 Tree temp =
   new Tree(2, tree.right.right, null);
 tree.right.right = null;
 tree.right = temp;
}
```



Solution Idea (by example)

- Consider what changes a procedure can make

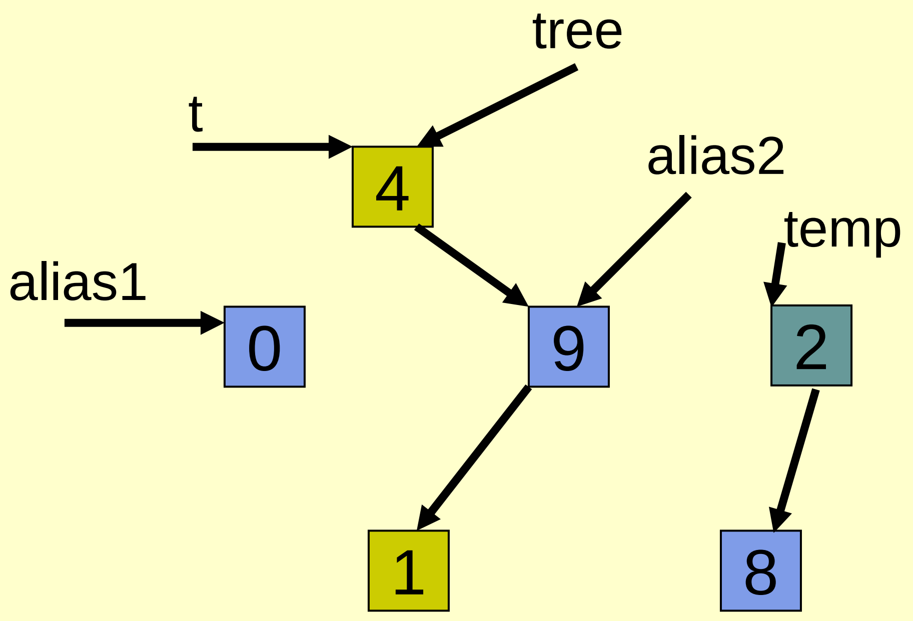
```
foo(t); ...

void foo (Tree tree) {
 tree.left.data = 0;
 tree.right.data = 9;
 tree.right.right.data = 8;
 tree.left = null;
 Tree temp =
   new Tree(2, tree.right.right, null);
 tree.right.right = null;
 tree.right = temp;
}
```



Solution Idea (by example)

- Consider what changes a procedure can make

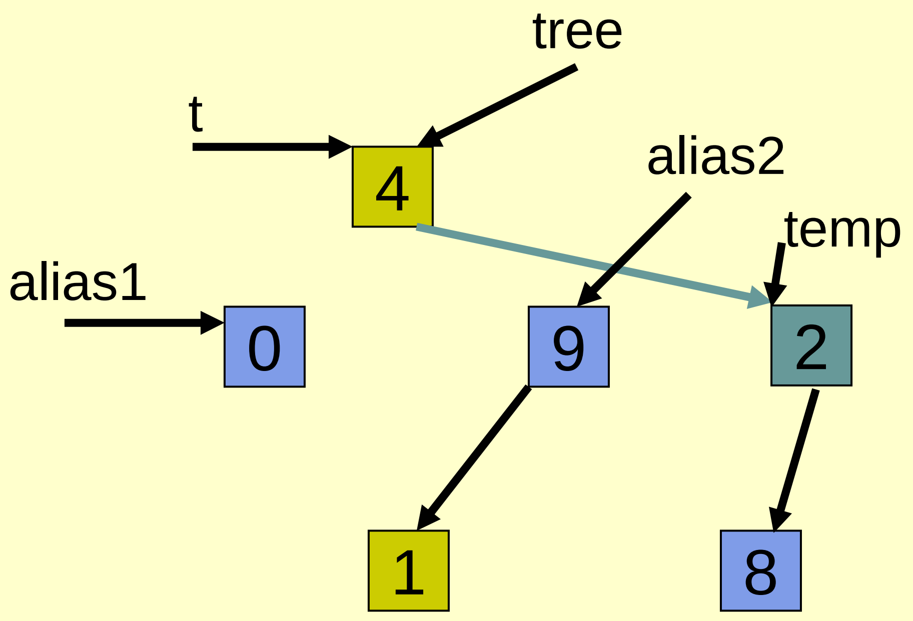
```
foo(t); ...

void foo (Tree tree) {
 tree.left.data = 0;
 tree.right.data = 9;
 tree.right.right.data = 8;
 tree.left = null;
 Tree temp =
   new Tree(2, tree.right.right, null);
 tree.right.right = null;
 tree.right = temp;
}
```



Solution Idea (by example)

- Consider what changes a procedure can make

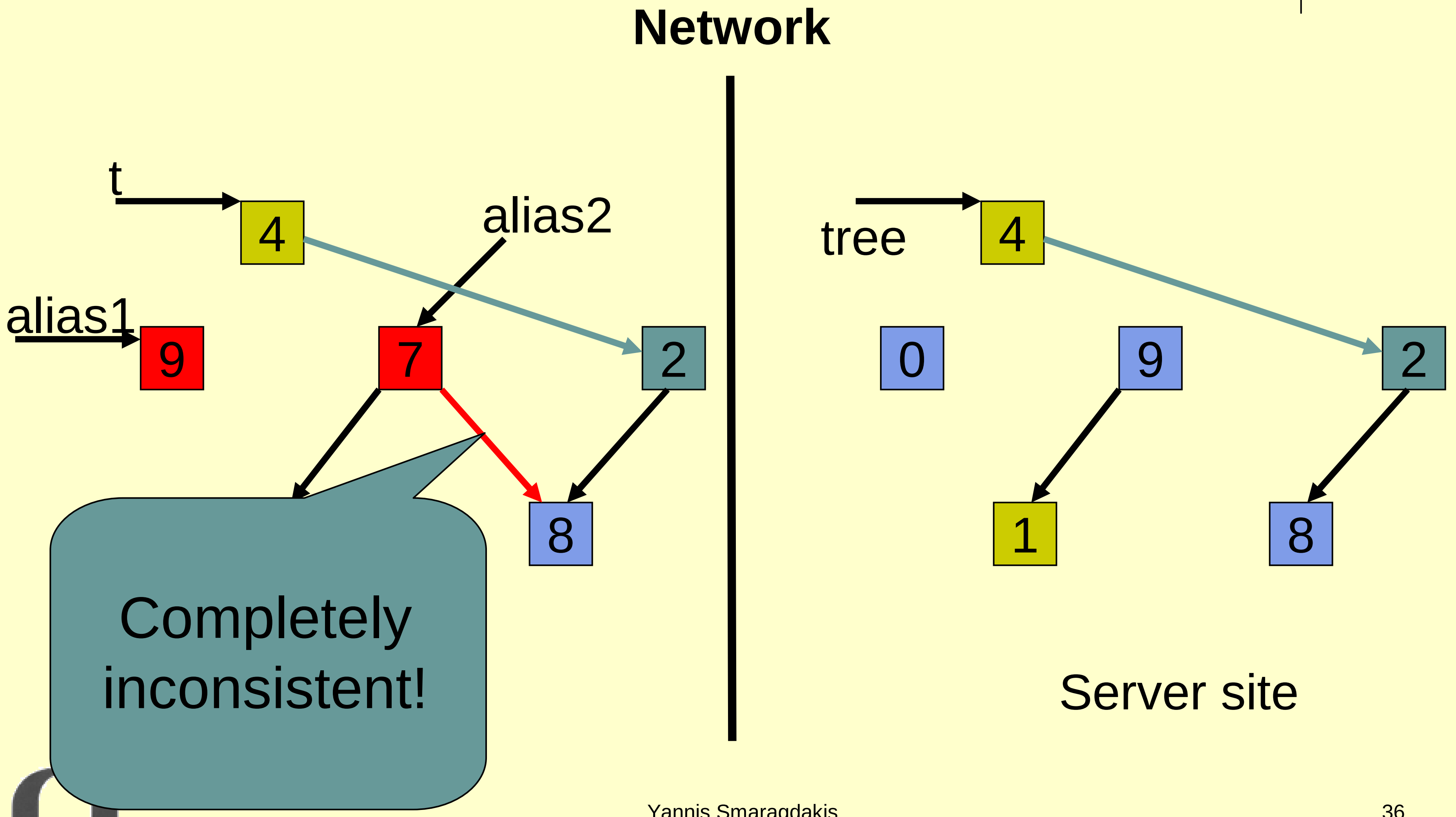
```
foo(t); ...

void foo (Tree tree) {
 tree.left.data = 0;
 tree.right.data = 9;
 tree.right.right.data = 8;
 tree.left = null;
 Tree temp =
   new Tree(2, tree.right.right, null);
 tree.right.right = null;
 tree.right = temp;
}
```



Previous Attempts: DCE RPC

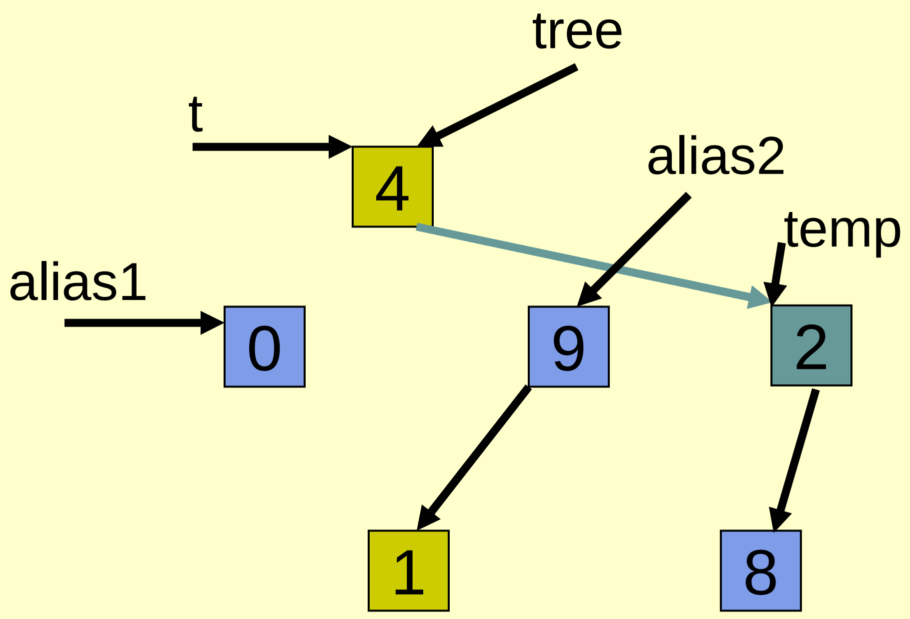
- DCE RPC is the foremost example of a middleware design that supports restoring remote changes
- The most widespread DCE RPC implementation is Microsoft RPC (the base of middleware for the Microsoft operating systems)
- Supports “full pointers” (ptr) which can be aliased
- No true copy-restore: aliases not correctly updated
  - for complex structures, it’s not enough to copy back and restore the value of arguments

DCE RPC: stops short!

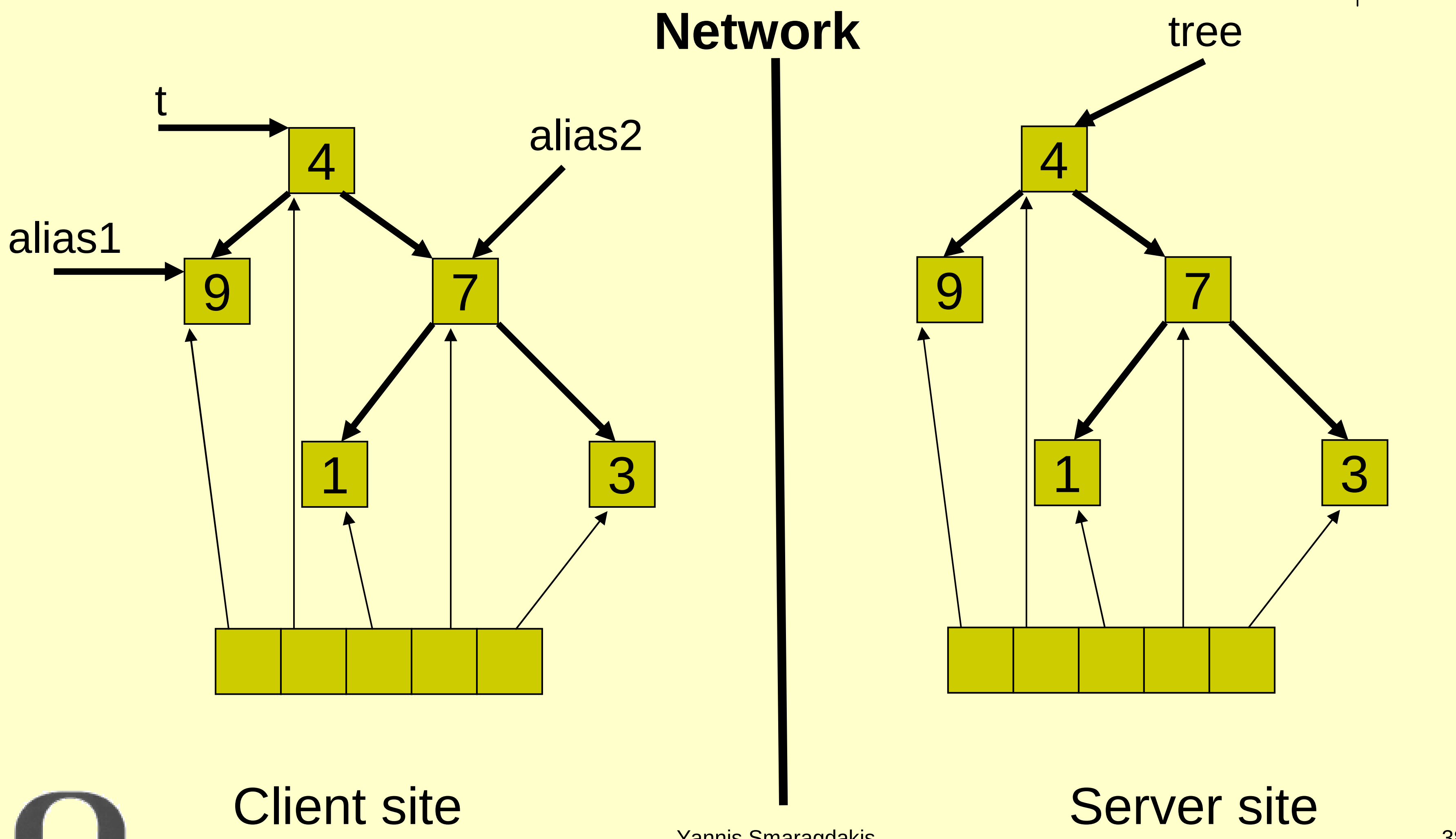


Solution Idea (by example)

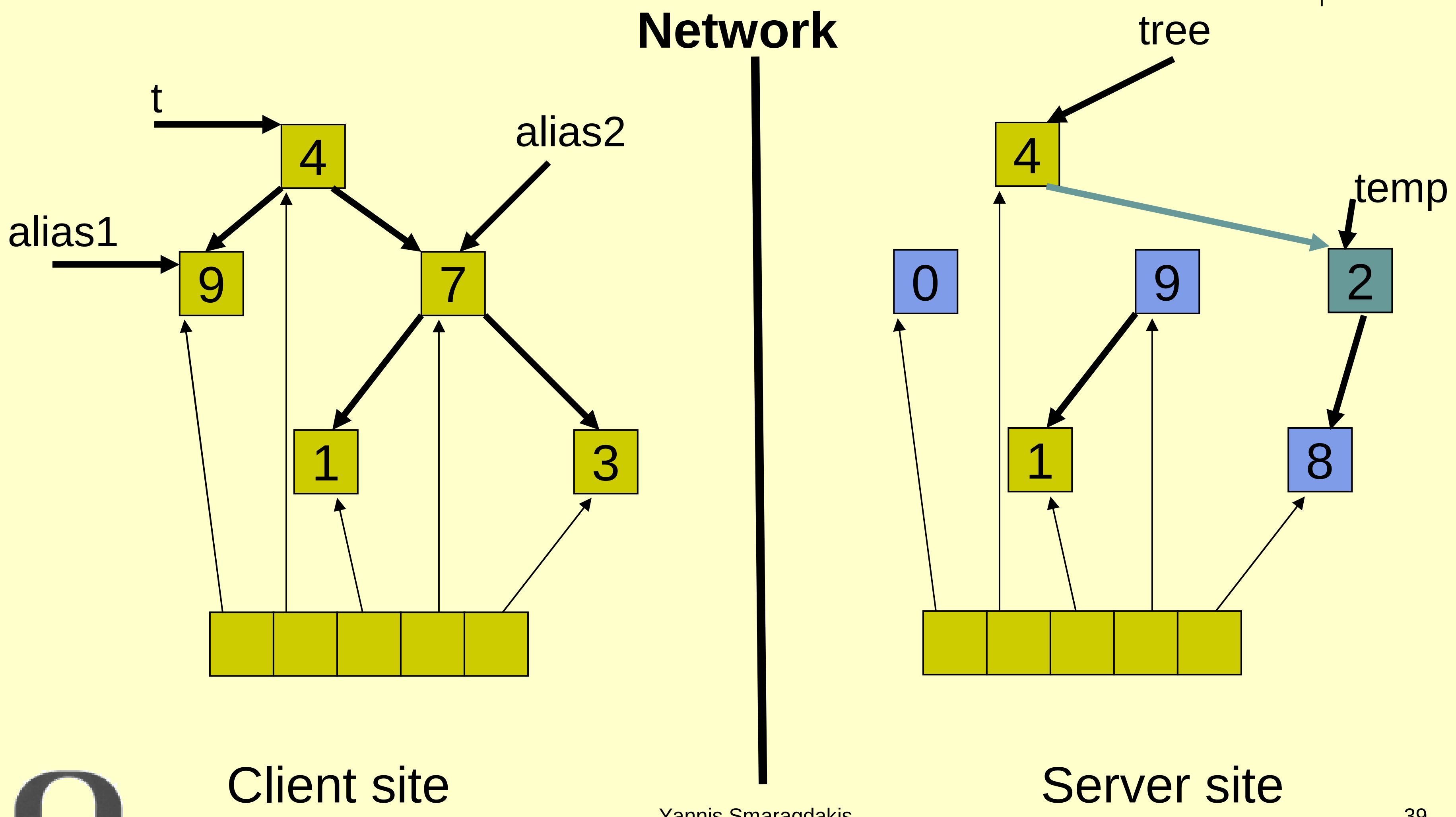
- Key insight: *the changes we care about are all changes to objects reachable from objects that were originally reachable from arguments to the call*
- Three critical cases:
  - changes may be made to data now unreachable from t, but reachable through other aliases
  - new objects may be created and linked
  - modified data may now be reachable only through new objects



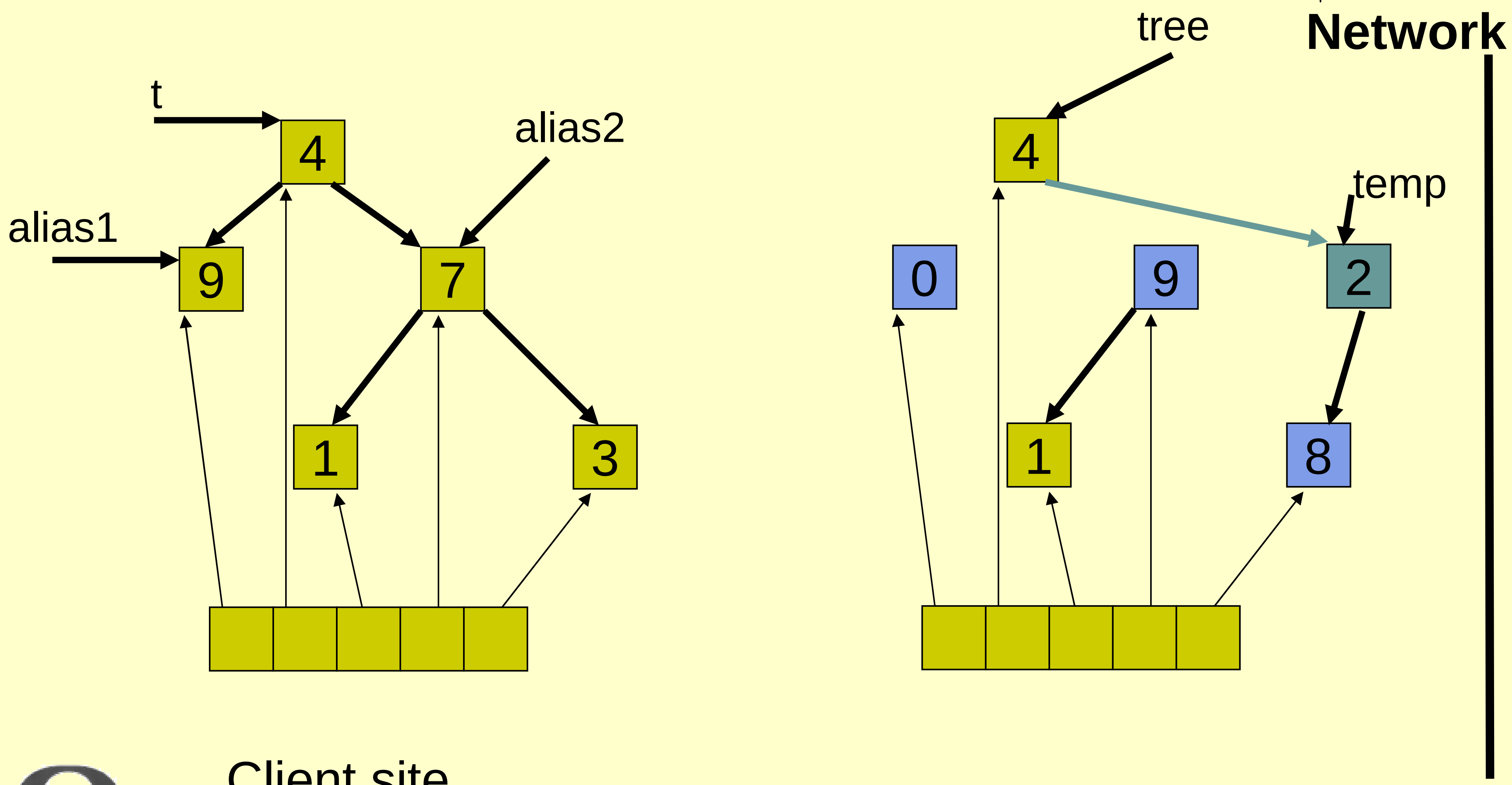
NRMI Algorithm (by example): identify all reachable



Algorithm (by example): execute remote procedure



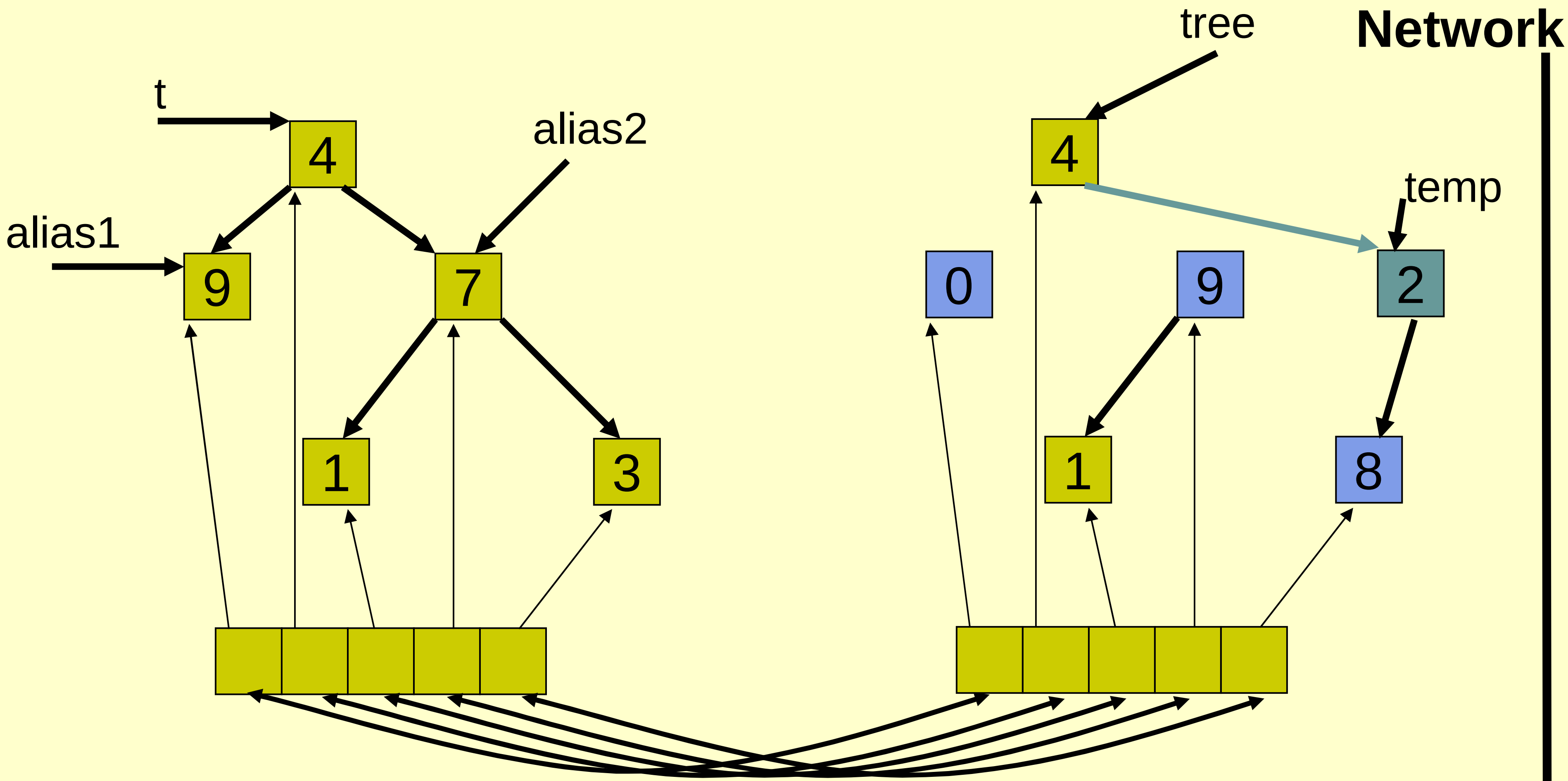
Algorithm (by example): send back all reachable



Client site

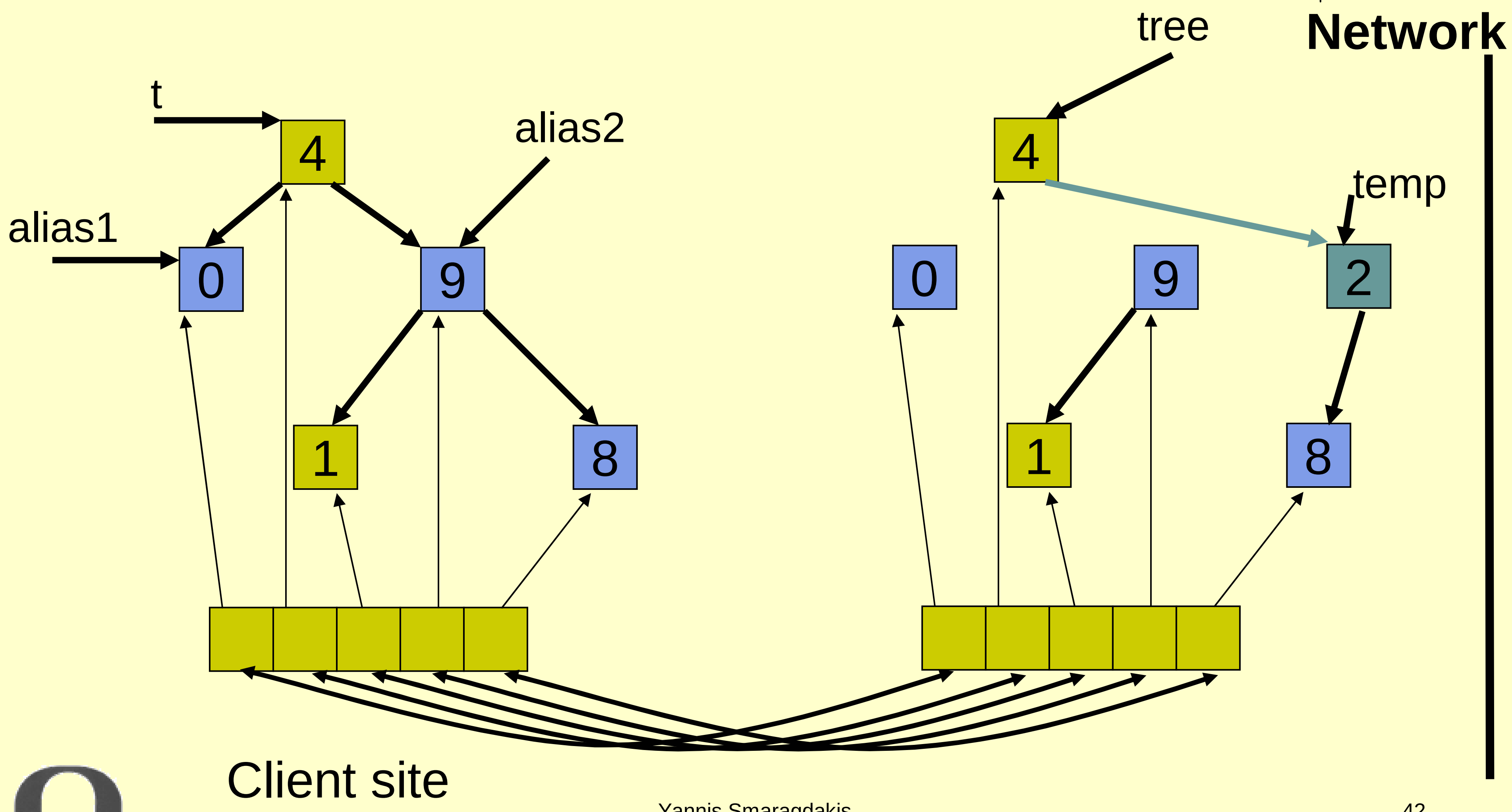
Algorithm (by example): match reachable maps

Network

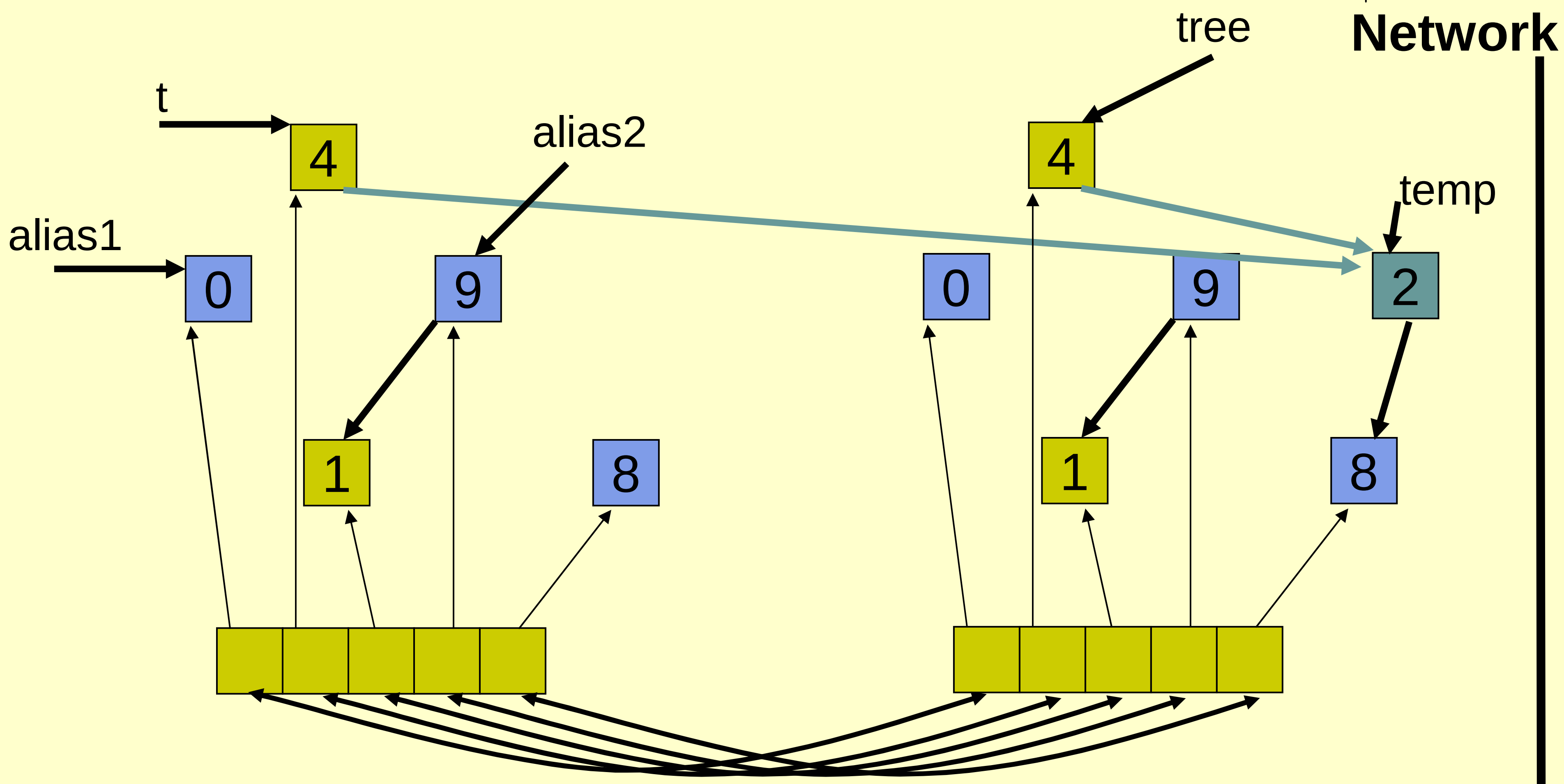


Client site

Algorithm (by example): update original objects

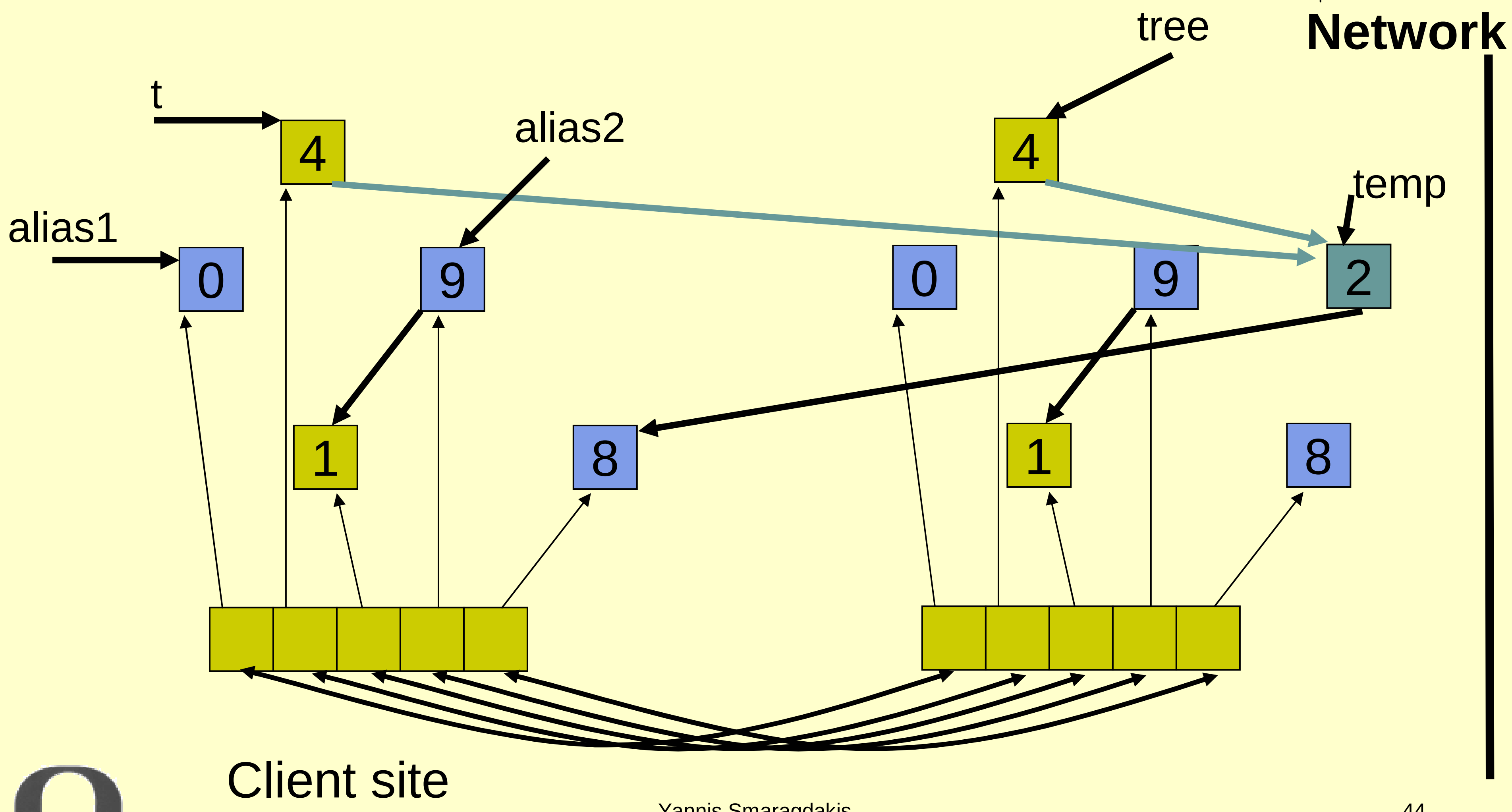


Algorithm (by example): adjust links out of original objects

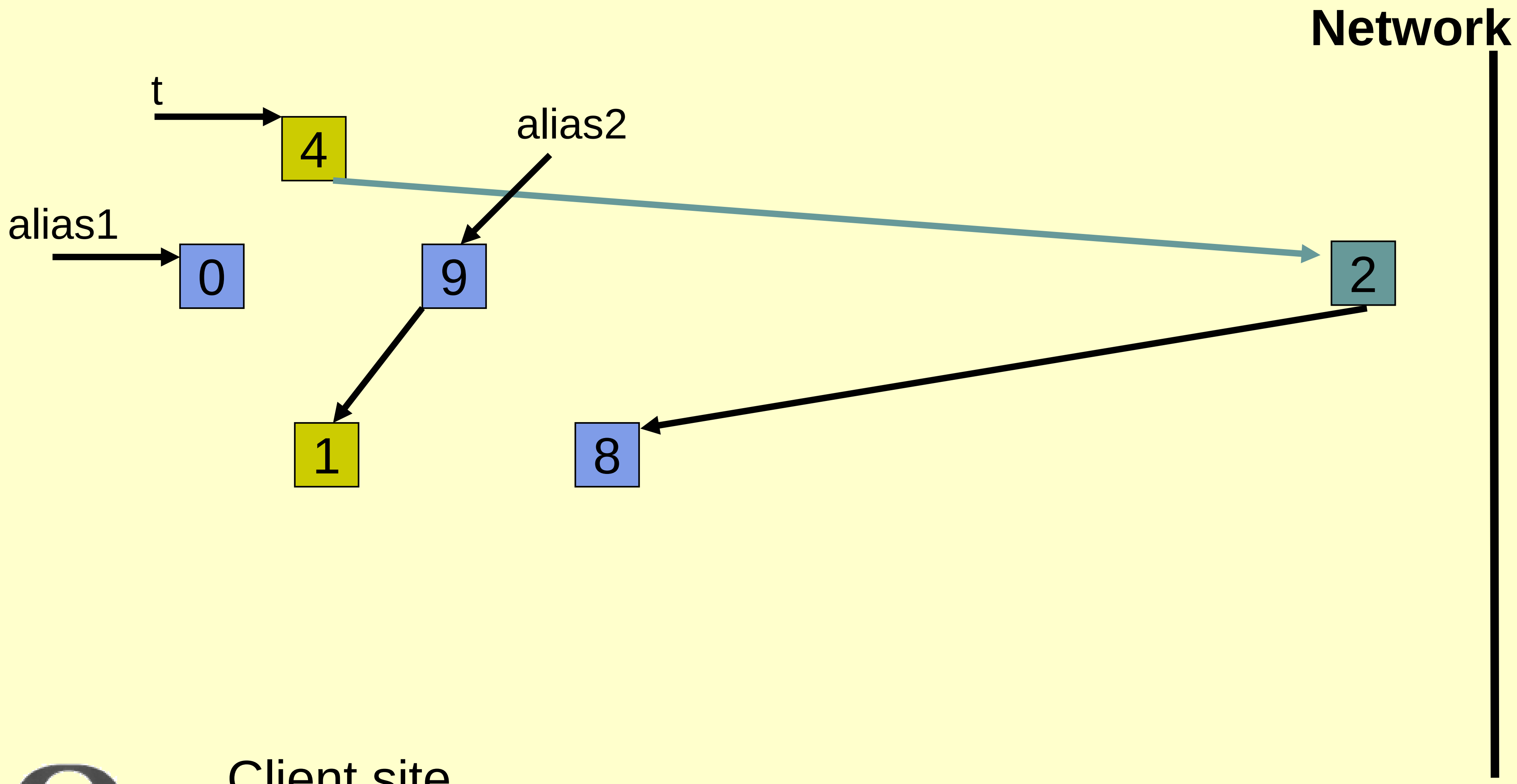


Client site

Algorithm (by example): adjust links out of new objects



Algorithm (by example): garbage collect



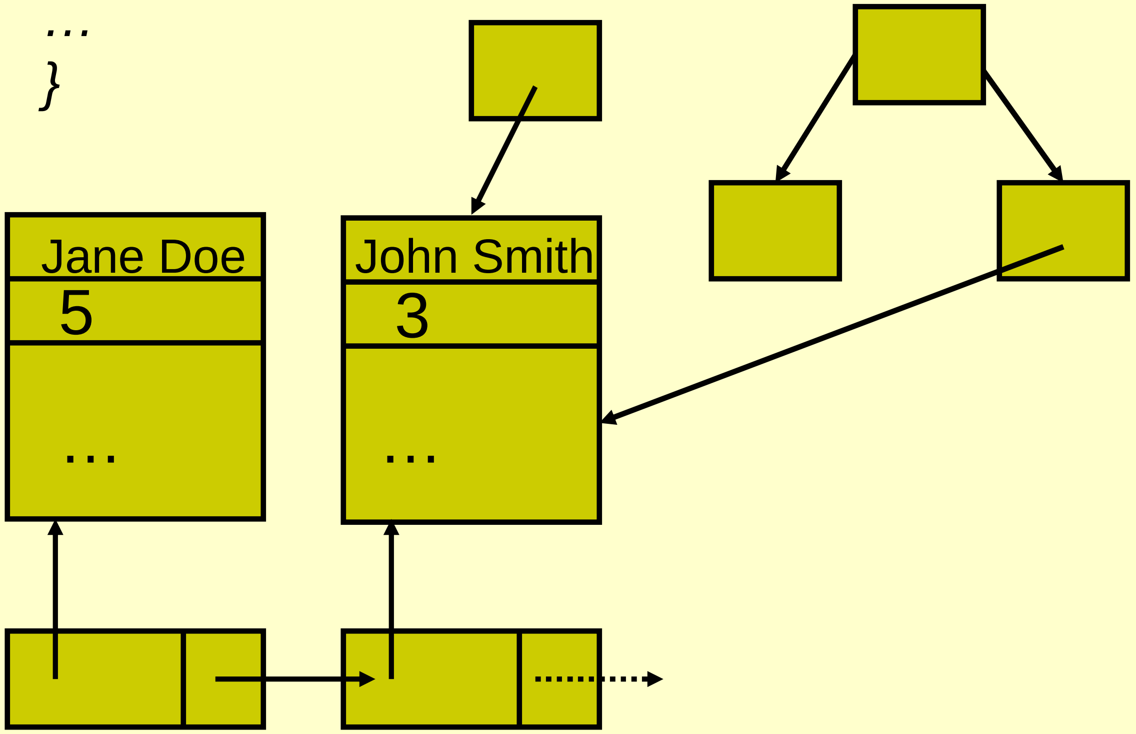
Client site

Usability and Performance

- NRMI makes programming easier
  - no need to even know aliases
  - even if all known, eliminates many lines of code (~50 per remote call/argument type—26% or more of the program for our benchmarks)
  - common scenarios:
    - GUI patterns like MVC: many views alias same model
    - multiple indexing (e.g., customers + transactions crossreferenced)

Example (Multiple Indexing)

class Customer {
String name;
int orders;
…
}



Network

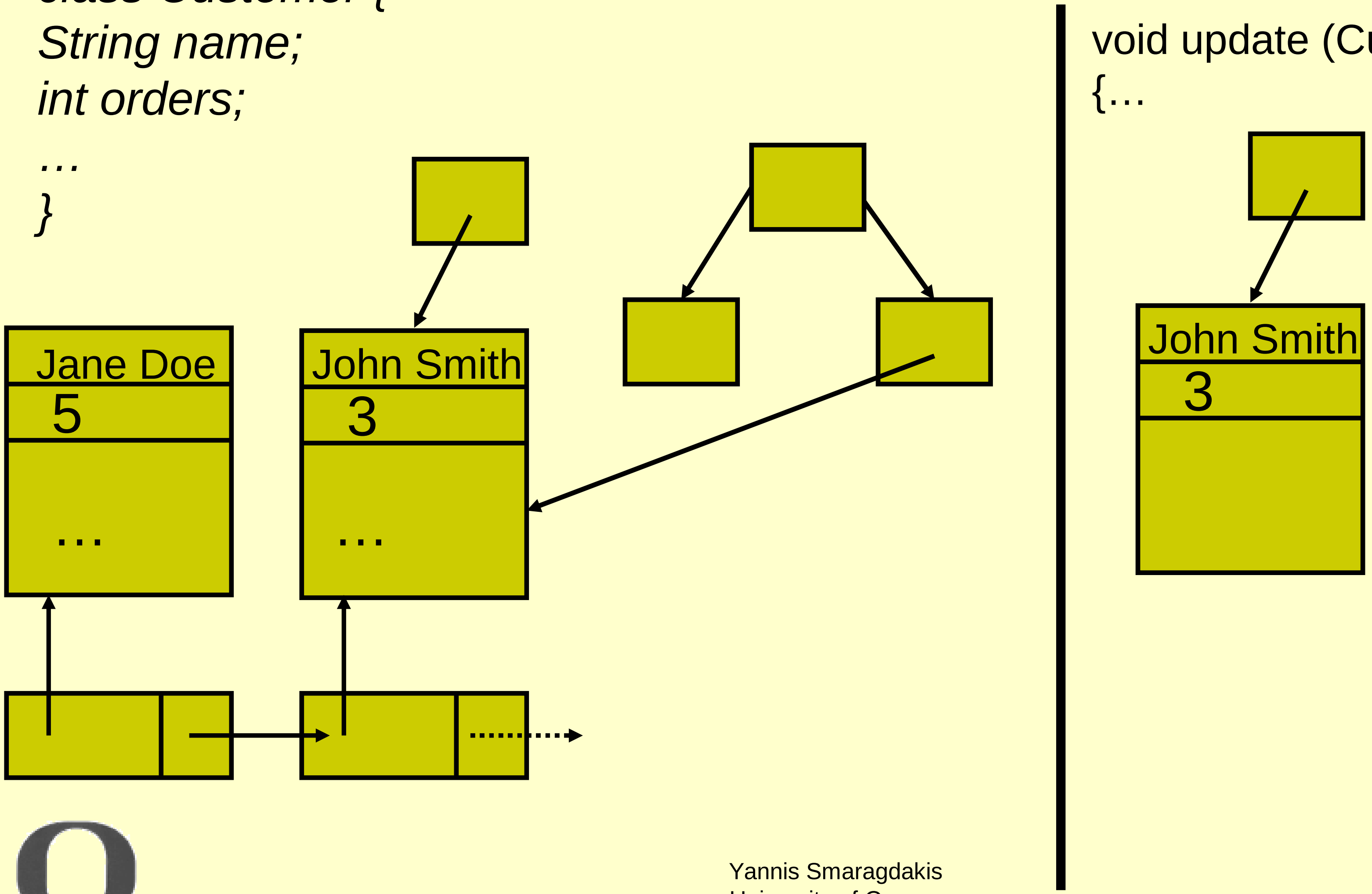
void update (Customer c)
{…

Example (Multiple Indexing)

class Customer {
String name;
int orders;
…
}

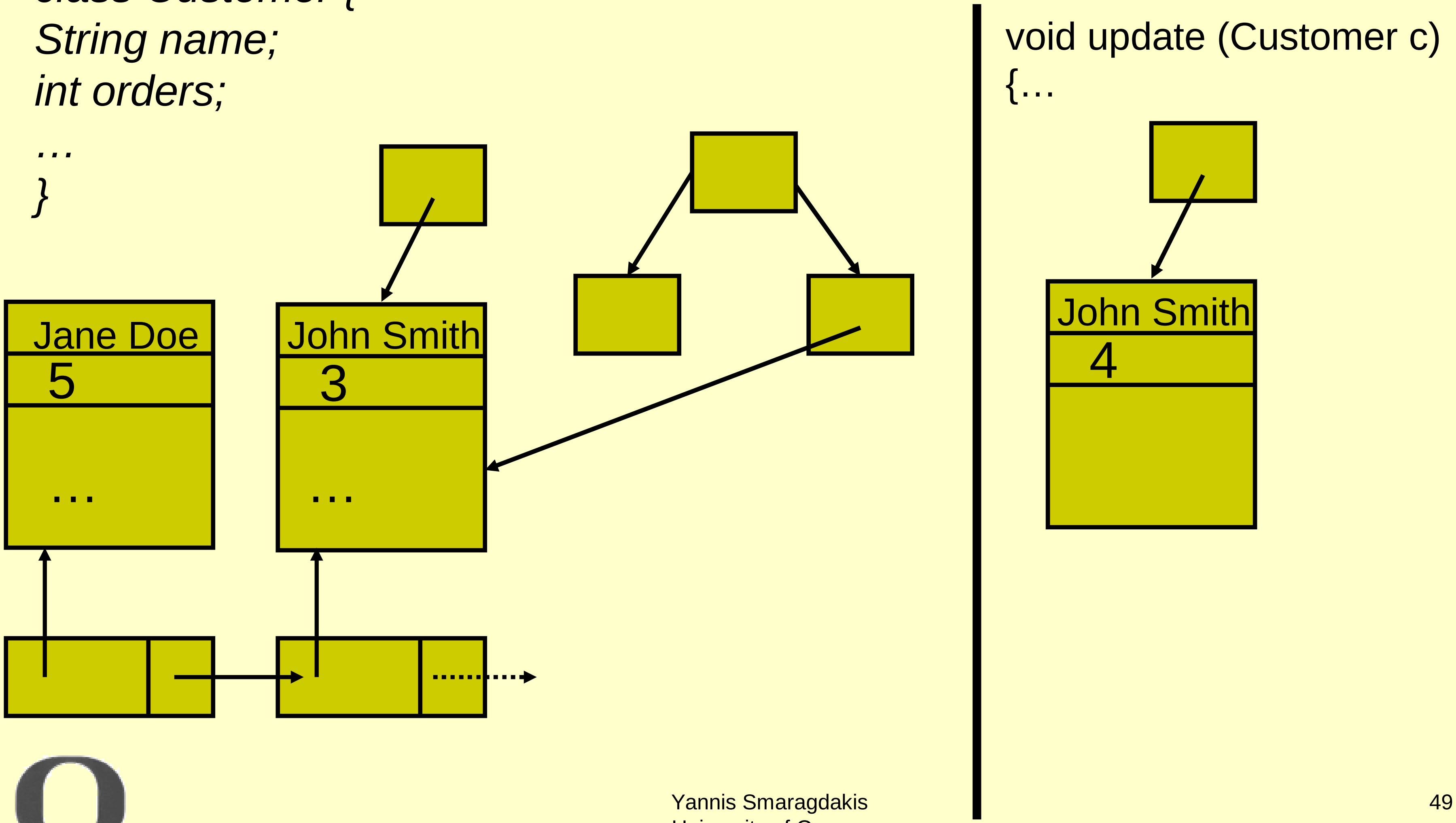
Network

void update (Customer c)
{…



Example (Multiple Indexing)

class Customer {
String name;
int orders;
…
}



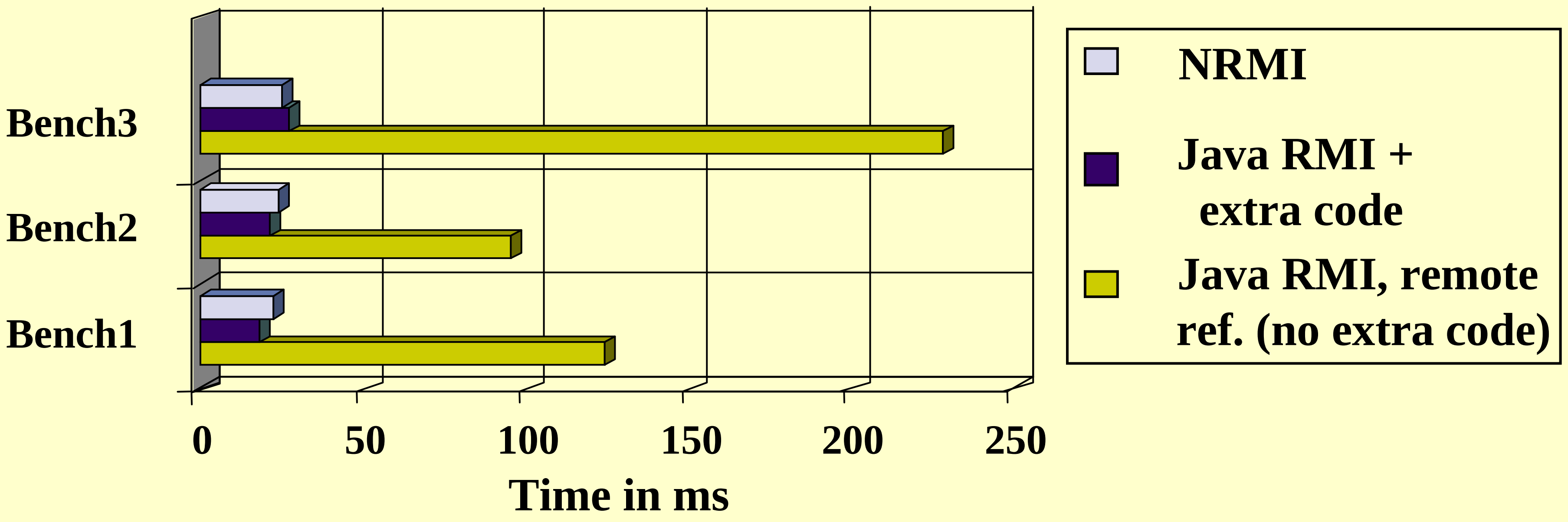
void update (Customer c)
{…

Performance

- We have a highly optimized implementation
  - algorithm implemented by tapping into existing serialization mechanism, optimized with Java 1.4+ “unsafe” facility for direct memory access

Experimental Results

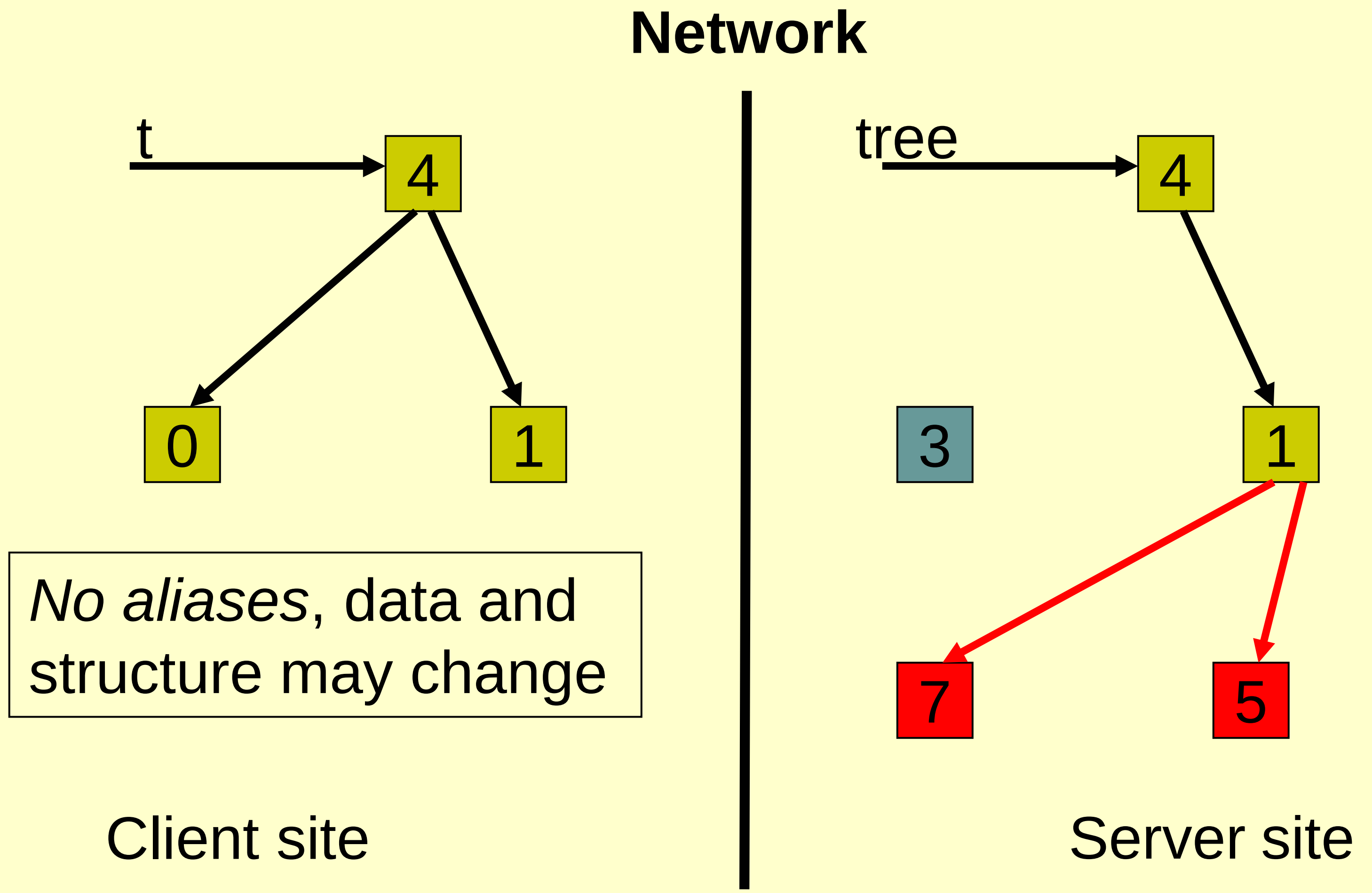
Tree of **256** nodes



Benchmarks

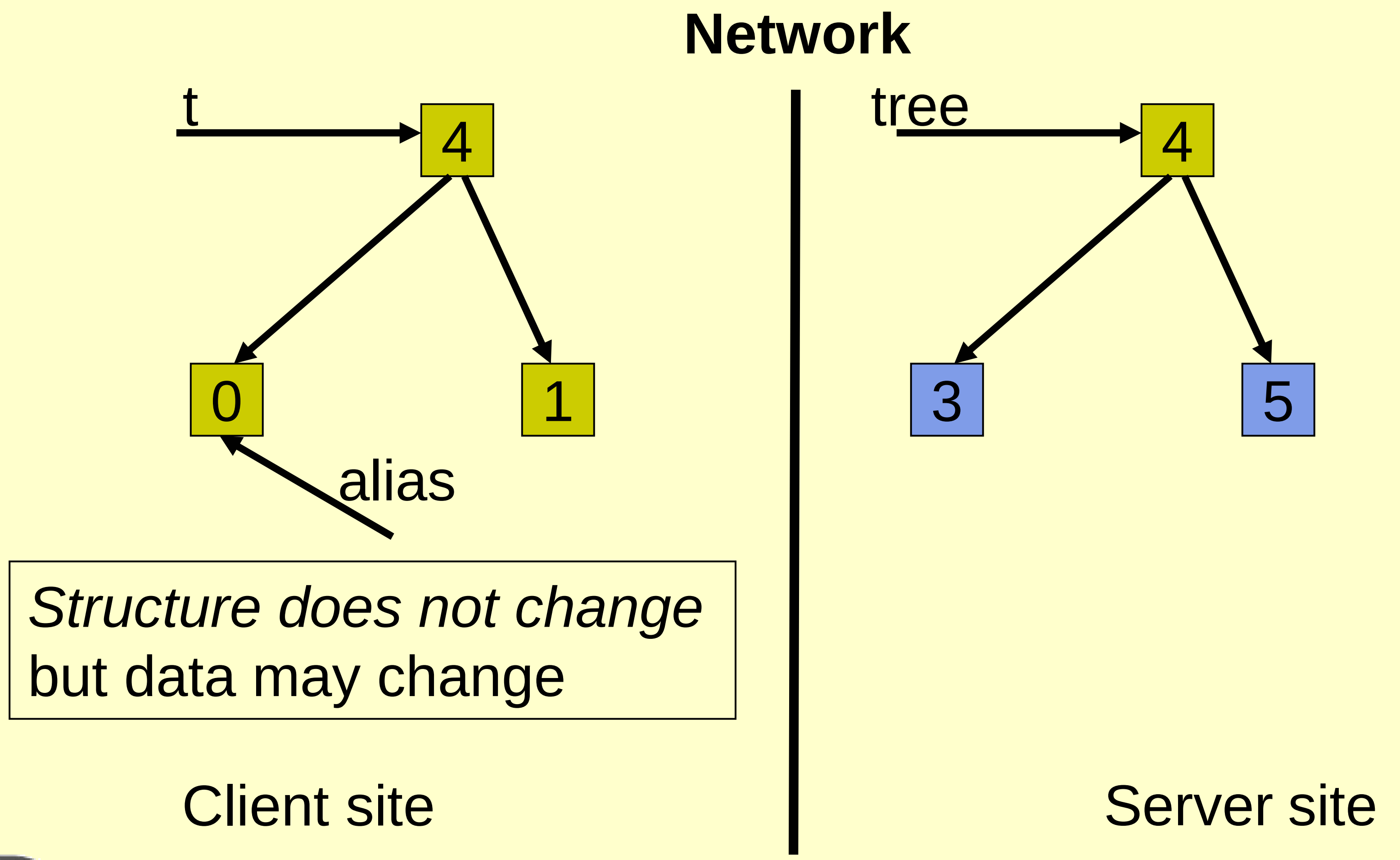
- Each benchmark passes a single randomly-generated binary tree parameter to a remote method
- Remote method performs random changes to its input tree
- We try to emulate the ideal a human programmer would achieve
- *The invariant maintained is that all the changes are visible to the client*

Benchmark Scenario 1



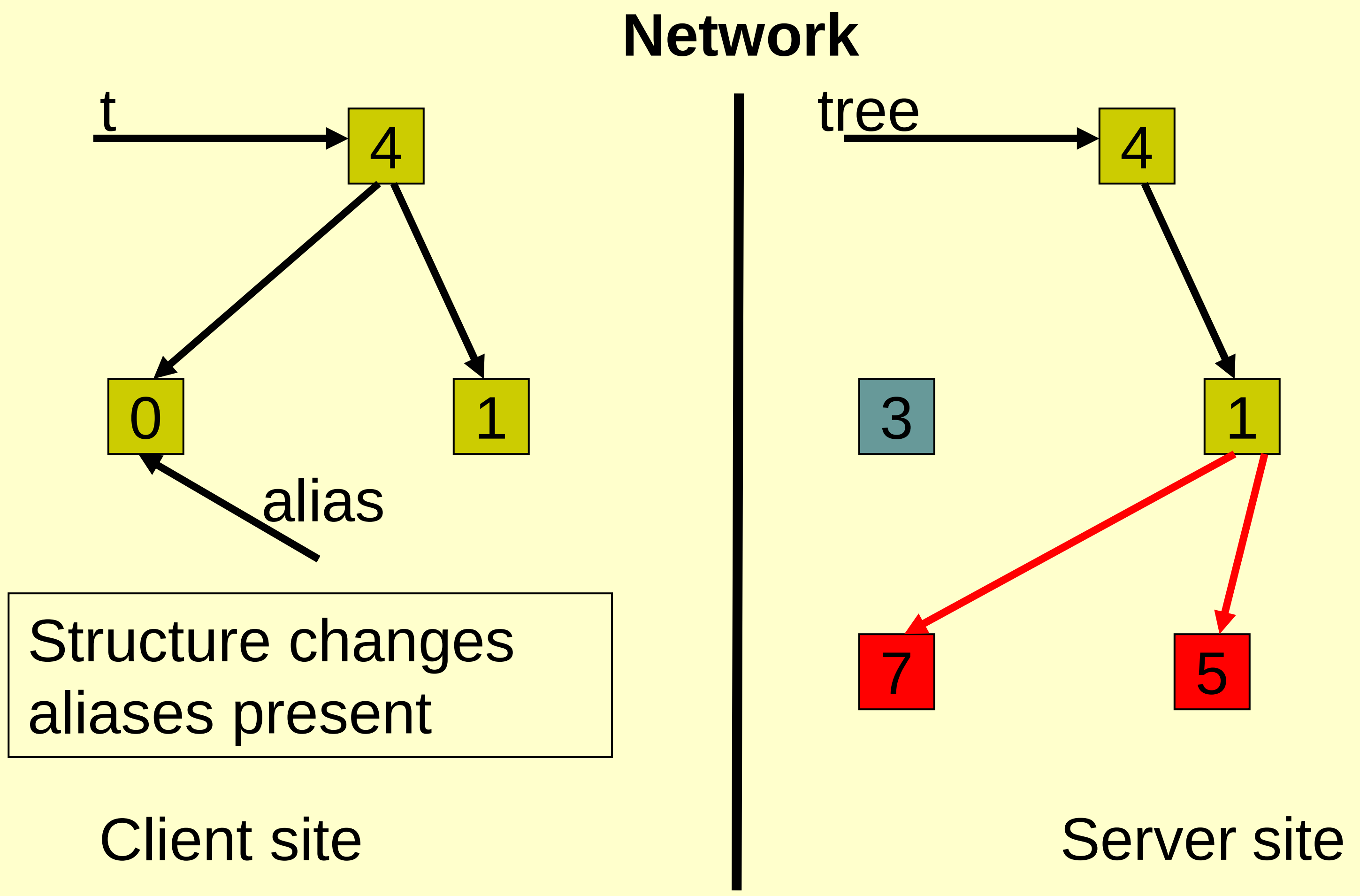
No aliases, data and structure may change

Benchmark Scenario 2



Structure does not change
but data may change

Benchmark Scenario 3



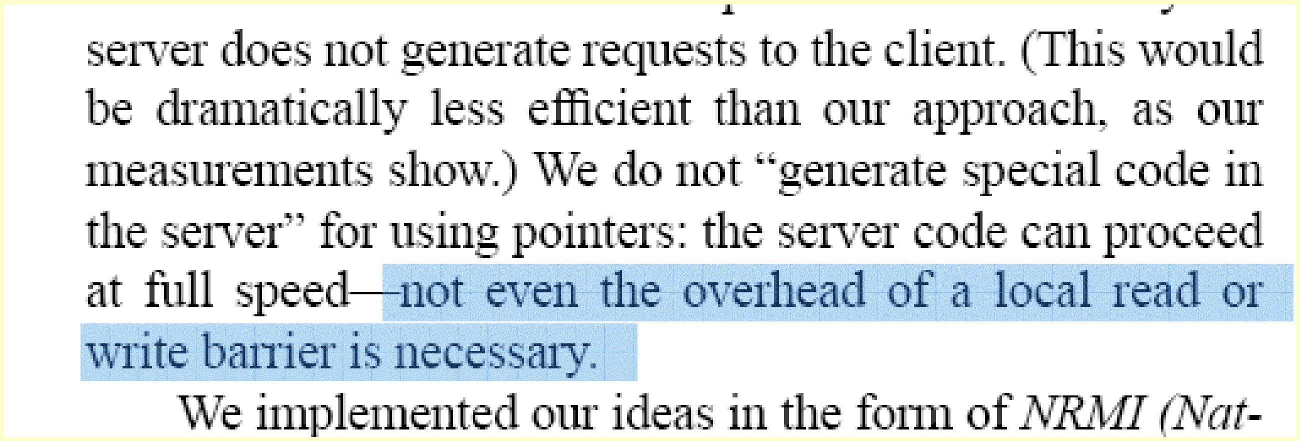
Structure changes
aliases present

Higher-level Distributed Programming Facilities

- NRMI is a medium-level facility: it gives the programmer full control, imposes requirements
  - good for performance and flexibility
  - low automation
- For single threaded clients and stateless servers, NRMI semantics is (provably) identical to local procedure calls
  - but statelessness is restrictive
- There are higher-level models for programming distributed systems
  - the higher the level, the more automation
  - the higher the level, the smaller the domain of applicability

Retrospective: What Helped Solve the Problem?

- An instance of “looking at things from the right angle”
  - a languages background helped a lot:
    - with defining precisely what copy-restore means
    - with identifying the key insight
    - with coming up with an efficient algorithm



server does not generate requests to the client. (This would be dramatically less efficient than our approach, as our measurements show.) We do not “generate special code in the server” for using pointers: the server code can proceed at full speed—not even the overhead of a local read or write barrier is necessary.

We implemented our ideas in the form of *NRMI (Nat-*

In Summary

What did I talk about?

This Talk

- NRMI: middleware offering a natural programming model for distributed computing
  - solves a long standing, well-known open problem!
- J-Orchestra: execute unsuspecting programs over a network, using program rewriting
  - led to key enhancements of a major open-source software project (JBoss)
- Morphing: a high-level language facility for safe program transformation
  - "bringing discipline to meta-programming"