Archipelago:
A Coordination-Oriented Measurement Infrastructure

Young Hyun
CAIDA

UCSD Syslunch
Apr 11, 2007

Outline

- background
- goals
- architecture
- examples
- status

Background

- Macroscopic Topology Project at CAIDA
  - represents our main effort in **active** network measurement
  - more than 8 years of data collection
  - running *skitter* on 20-25 “monitors” worldwide
  - > 12.7 billion complete *skitter* traces (as of Apr 2007)
  - CAIDA has used data for
    - AS graph poster
    - AS ranking
    - Internet Topology Data Kit (ITDK)
    - various topology analyses

skitter Measurement Infrastructure

central server

(CAIDA in San Diego)

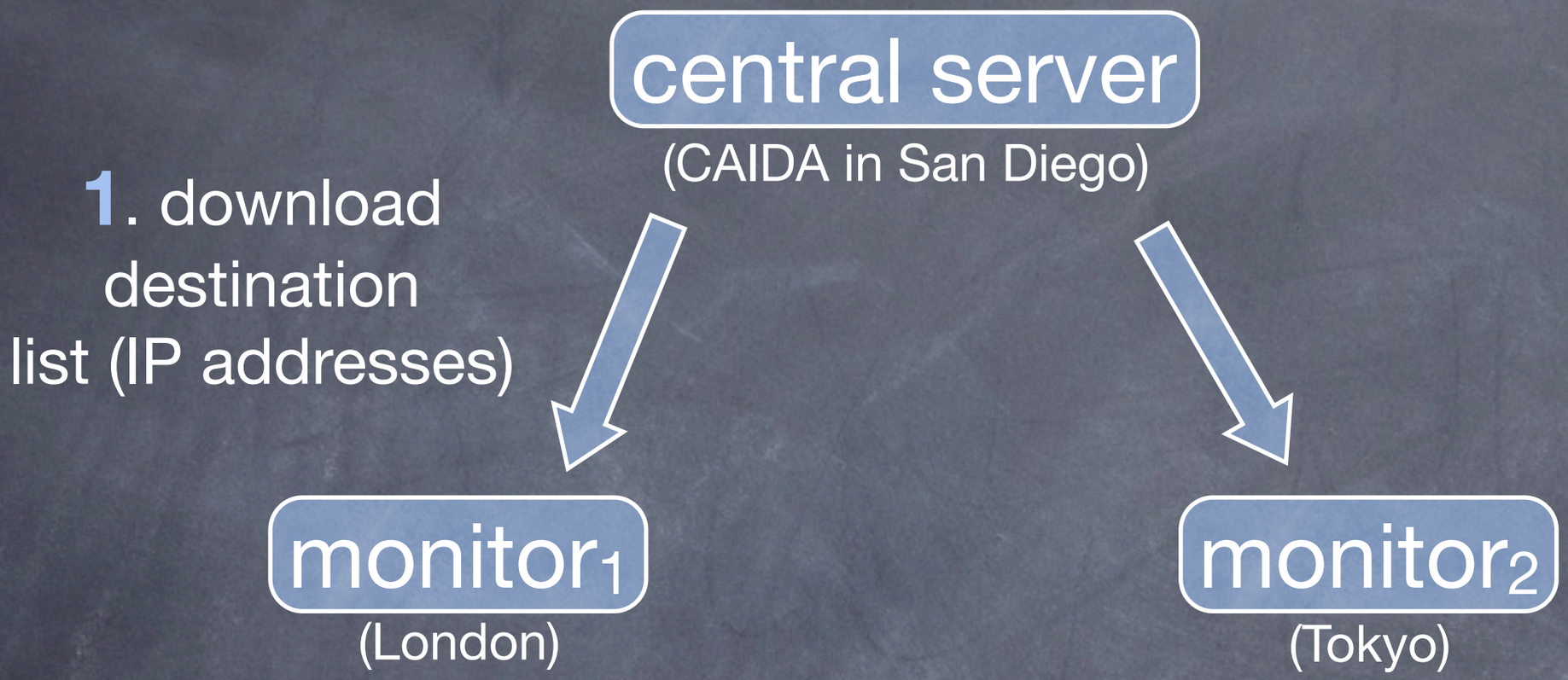
monitor$_1$

(London)

monitor$_2$

(Tokyo)

skitter Measurement Infrastructure



skitter Measurement Infrastructure

central server

(CAIDA in San Diego)

2. perform traceroute measurement

monitor$_1$

(London)

monitor$_2$

(Tokyo)

host$_1$

(US)

host$_2$

(Africa)

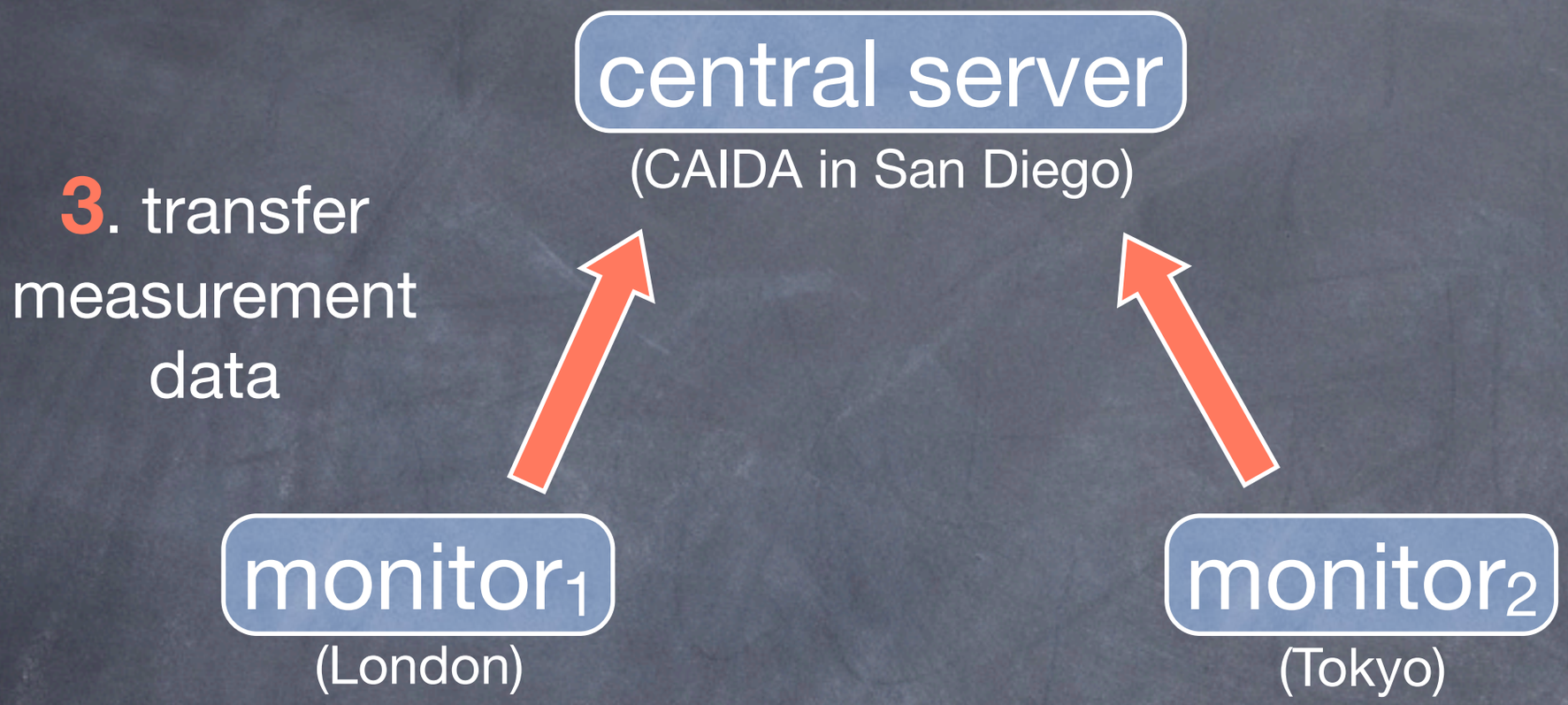
router

(Europe)

web server

(Oceania)

skitter Measurement Infrastructure



Introduction

- **Archipelago** (Ark) is CAIDA's next generation active measurement infrastructure
- Ark is an upgrade to skitter infrastructure
  - replacing software
    - using **scamper** instead of skitter for taking measurements
      - IPv4; IPv6; ICMP, UDP, and TCP traceroute and ping; Paris traceroute; path MTU discovery
    - using new Ark software for communication, management, security, etc.
  - adding/upgrading hardware
    - adding several dozen monitors to infrastructure
    - deploying monitors in 20 countries that never had a monitor before

Introduction

- Ark is an *infrastructure*, not a tool
  - concerned with system-level issues
    - security, data management, software distribution, communication, scheduling, ...
  - accommodates open-ended set of tools
    - traceroute, ping, one-way loss, bandwidth estimation, DNS performance, router alias resolution, ...
- *could* be used for passive measurement but geared toward active
  - passive measurement: simple, few locations, high data volume
  - active measurement: complex, highly distributed, low data volume

Goals

- a step toward a community-oriented measurement infrastructure
  - collaborators can run vetted measurements on security-hardened platform
  - general public can perform highly-restricted measurements
  - tailored for network measurement -- not broad-scope distributed experimental platform
    - inspired by PlanetLab but not PlanetLab

Goals

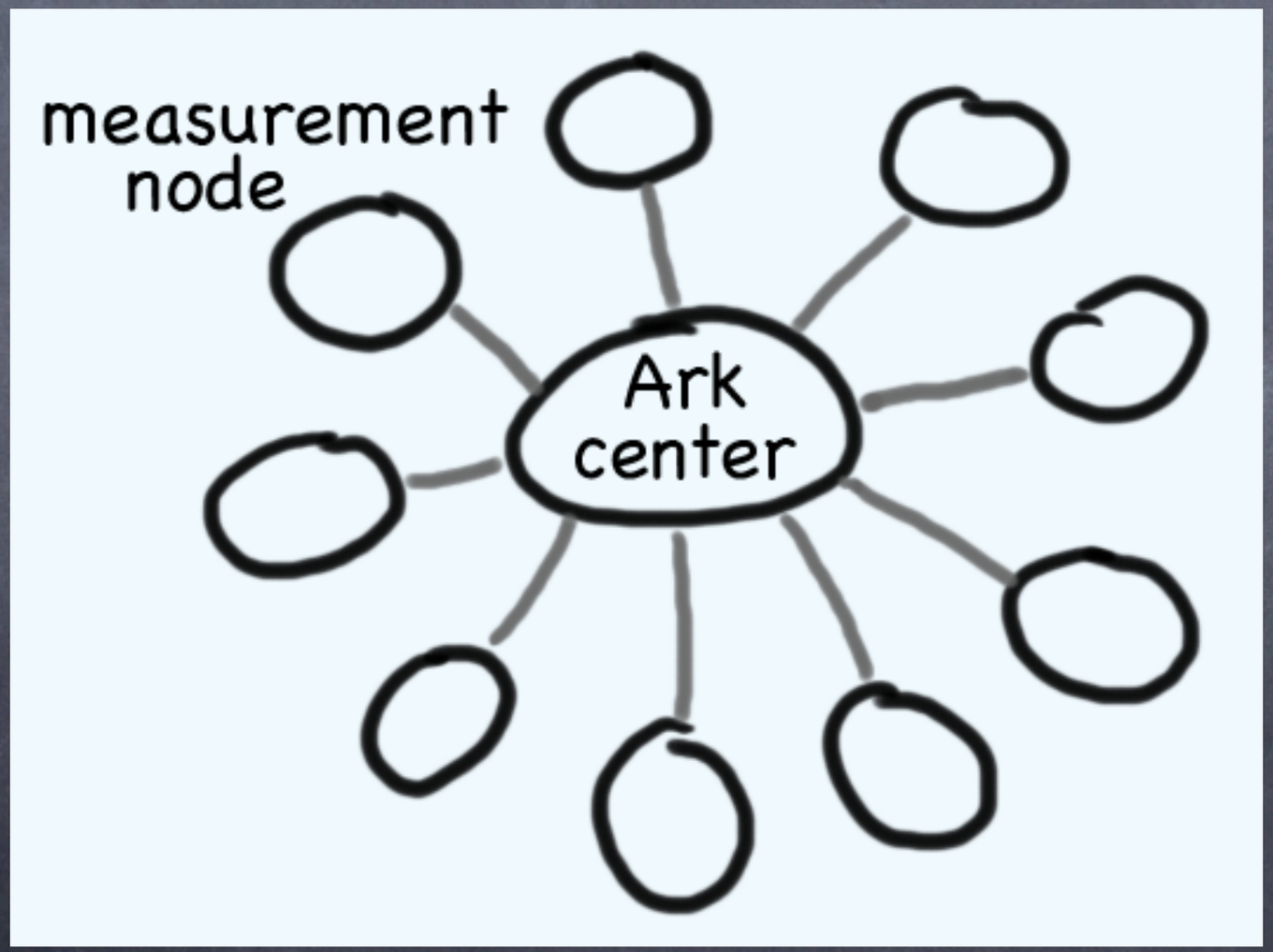
- greater scalability and flexibility
  - scalability in system management, monitor deployment, measurement efficiency, resource utilization
  - flexibility in measurement method, scheduling, data collection
- platform for measurement tool development, experimentation, deployment
  - raise level of abstraction with high-level API and scripting language
    - inspired by Scriptroute but not Scriptroute
  - factor out security, software distribution, data collection, etc. from tool development

Architecture

- topology
- security
- communication & coordination

Topology

- Ark is physically composed of measurement *nodes* (machines) located in various networks worldwide
  - measurement nodes connected to central server (at CAIDA) over Internet, forming a logical star topology



Architecture

- topology
- security
- communication & coordination

Security Features

- secure communication
- process isolation via sandboxing (FreeBSD jail)
- rate & resource limiting
- packet filtering
- fine-grained access control of resources

Security Features

- multiple levels of trust:
  - **stranger** (general public) -- no trust
    - no direct access to infrastructure; must access through, say, a web form
    - allow pre-defined set of restricted rate-limited measurements, similar to public traceroute servers
  - **acquaintance** -- low trust
    - direct access to infrastructure, but confined to sandbox
    - allow measurements based on granted privileges
    - subject to system and network resource limits
  - **collaborator** -- medium to high level of trust
    - direct access to infrastructure with optional restrictions

Security Model

- requirements
  - fine-grained authorization mechanisms for
    - reading and writing files
    - transferring measurement data and other files between hosts
    - accessing privileged or confidential resources (e.g., raw sockets, SNMP counters)
    - opening communication channels
    - installing, executing, and stopping measurement software
- scalability
- ability to delegate management
  - delegate authorization duties for a subset of nodes
  - allow hosting organization to set site-specific maximum privileges
    - e.g., nothing beyond traceroute
    - finer control than coarse configuration settings

Security Model

- chosen approach: capabilities
  - a *capability* is an unforgeable object reference combined with list of rights
  - **possession** of a capability is necessary and sufficient authorization
  - access is granted by passing capabilities from one process to another

Architecture

- topology
- security
- communication & coordination

Communication & Coordination

a measurement infrastructure is a **distributed system** with many components that **must work together** in complex ways **toward a common goal**

Communication & Coordination

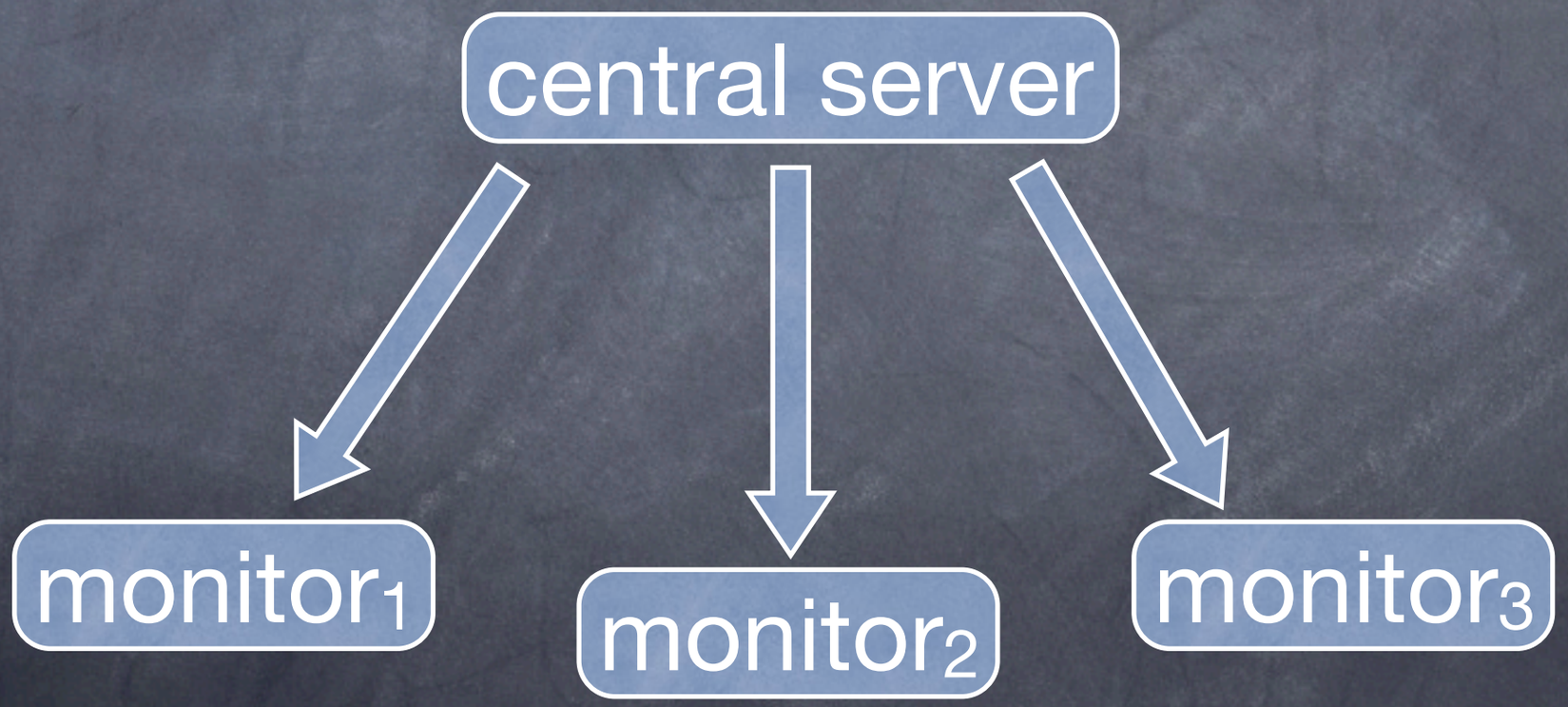
- **however**, a typical measurement infrastructure focuses only on:

central server

monitor$_1$ monitor$_2$ monitor$_3$

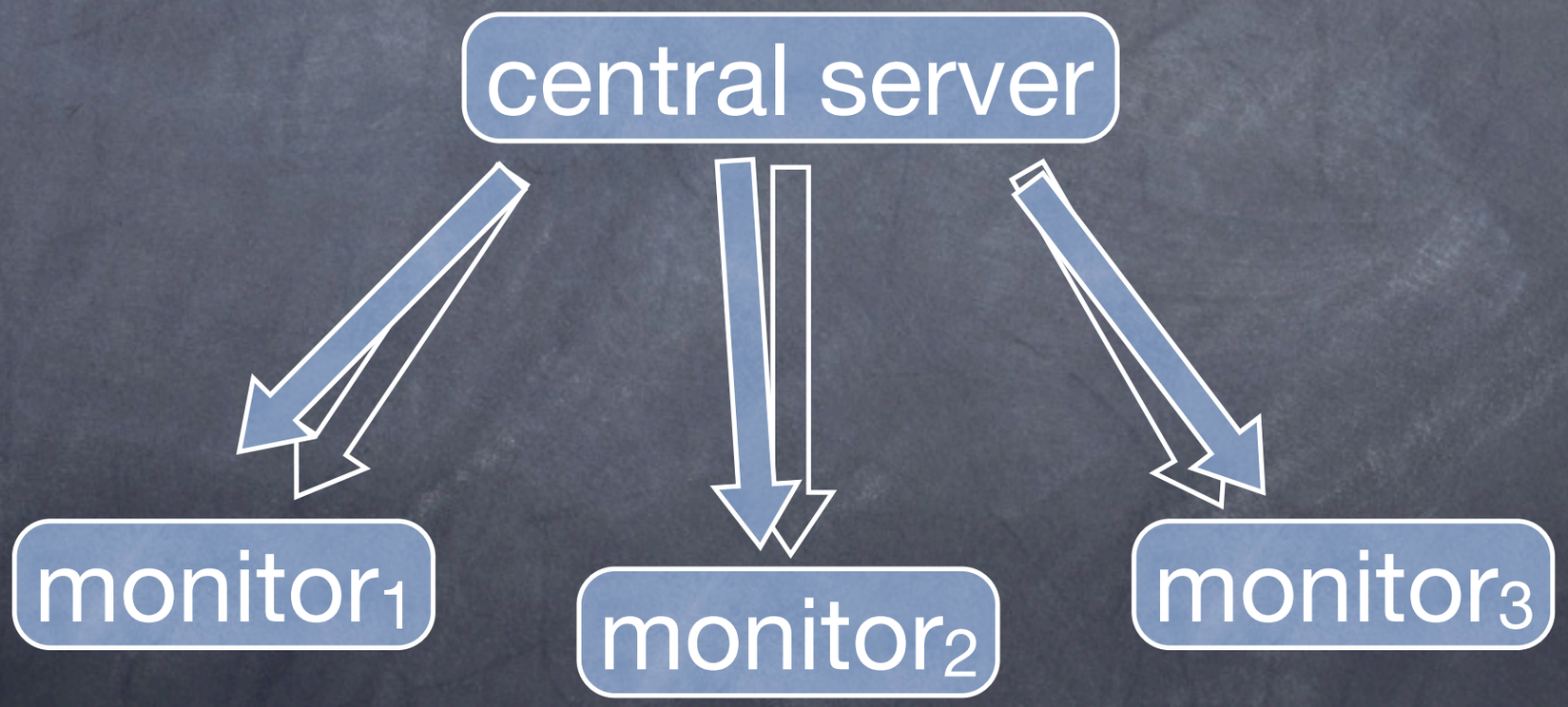
Communication & Coordination

- **however**, a typical measurement infrastructure focuses only on:
  - software deployment



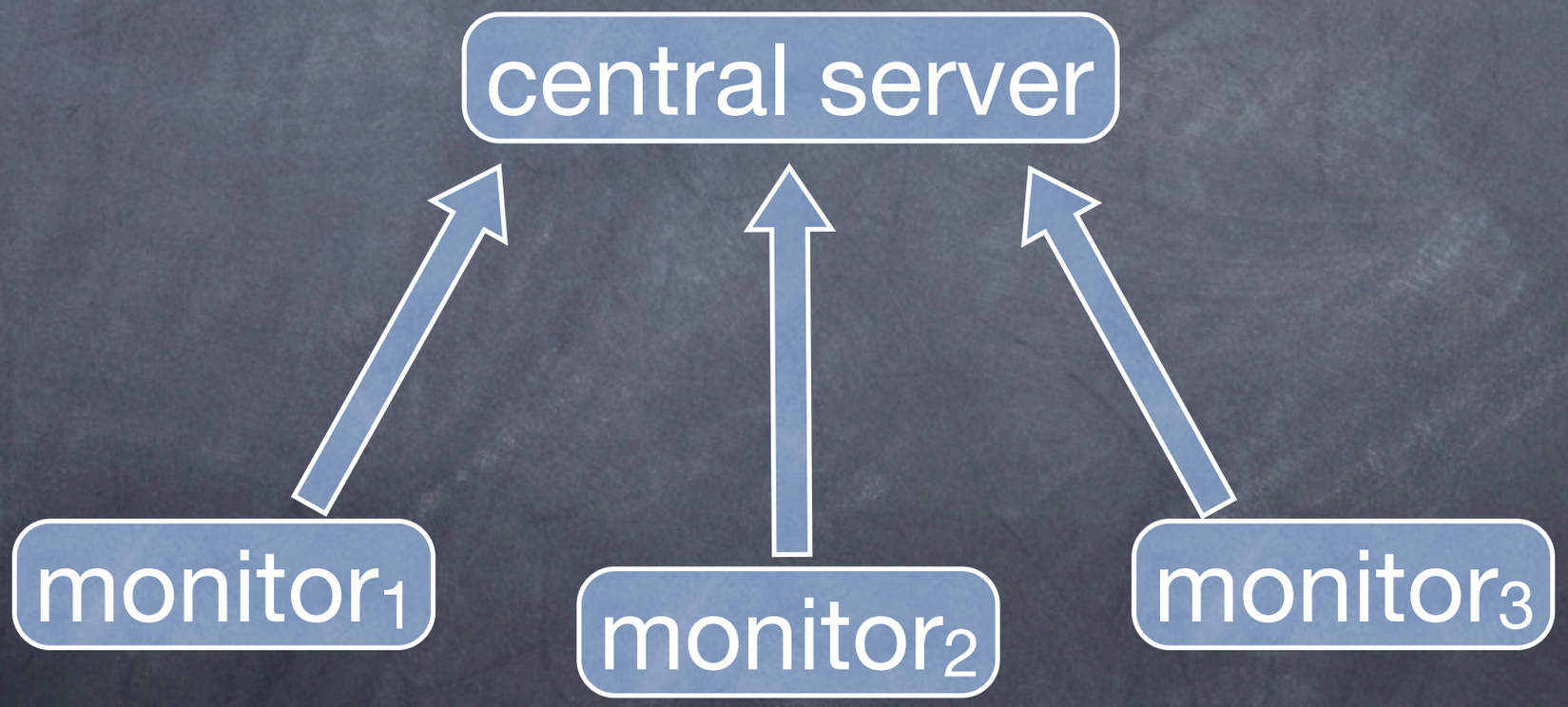
Communication & Coordination

- **however**, a typical measurement infrastructure focuses only on:
  - software deployment
  - measurement execution



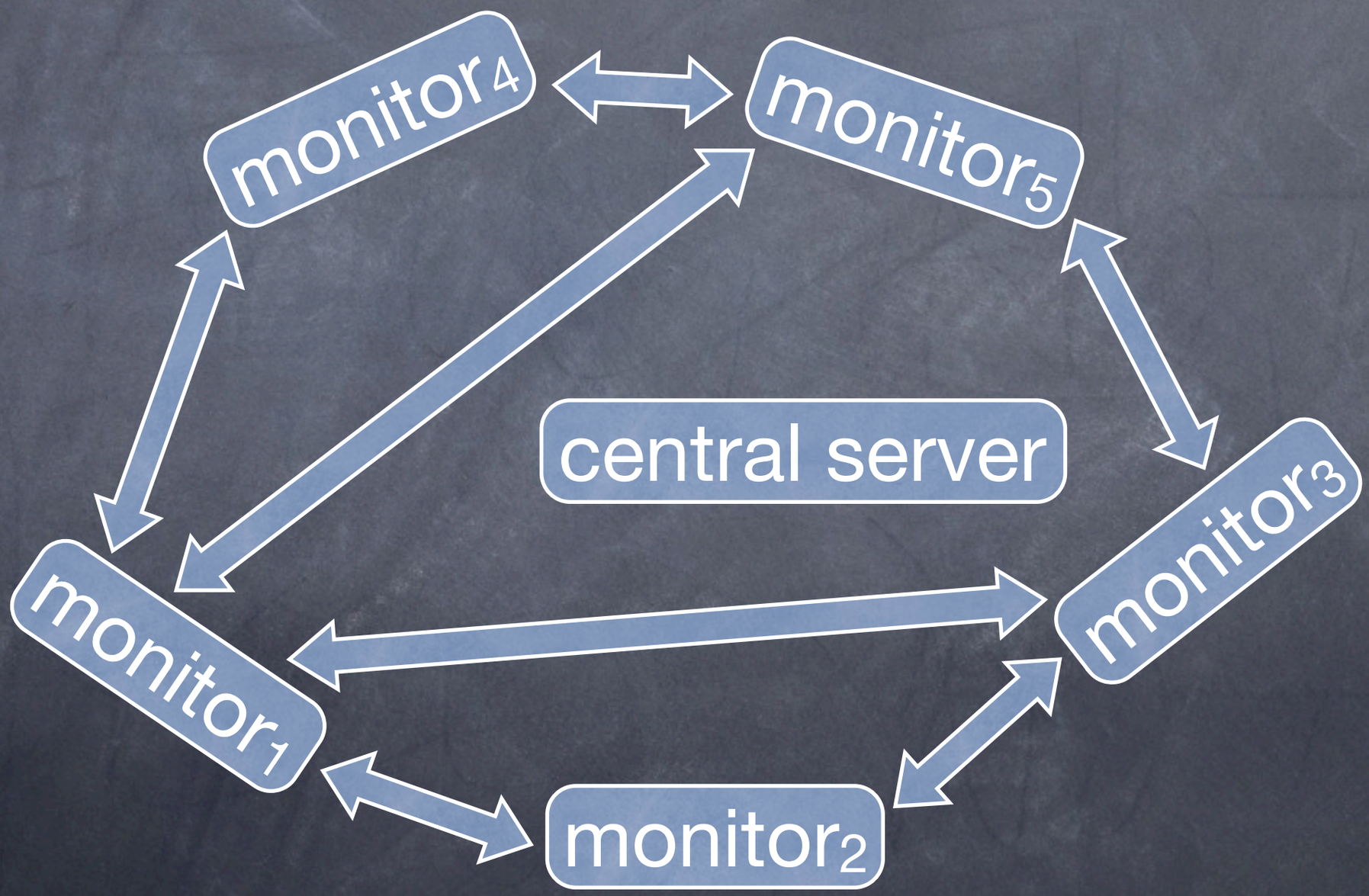
Communication & Coordination

- **however**, a typical measurement infrastructure focuses only on:
  - software deployment
  - measurement execution
  - data collection



Communication & Coordination

- missing out on a world of possibilities in decentralized **communication**, **interaction**, and **coordination**



Ark Vision

- **empower researchers** to use and build upon each other's work
  - similar to how the web allowed anyone to be a publisher, and changed everything
  - allow anyone to run a server that **provides a service**
  - decentralized -- no need to register; no need to install at central location

Communication & Coordination

- ability to communicate is **necessary but not sufficient**
- must go beyond communication to *coordination*

Communication & Coordination

- ability to communicate is **necessary but not sufficient**
- must go beyond communication to *coordination*
- coordination is about ...
  - scheduling
  - starting and stopping
  - controlling and guiding
  - satisfying dependencies and maintaining ordering
  - preparing for and cleaning up
  - distributing and collecting
- coordination is also important for collaborative use
  - to share and build upon each other's tools

Coordination Facility

- coordination is usually implemented in an **ad-hoc** manner on top of a communication facility
- general facility for directly implementing coordination is valuable
  - abstracts away programming details
  - lowers barrier to implementing remotely controllable components
  - easier to understand and verify correctness of coordinated behavior
  - easier to re-use or adapt *coordination patterns*

Coordination Facility

- Ark provides a ***tuple space*** for implementing coordination
  - tuple space is a **distributed shared memory** coupled with certain operations
  - tuple space originated in the Linda coordination language created in the mid-1980's by David Gelernter
    - further developed and refined over the years by researchers

Coordination Facility

- Ark provides a ***tuple space*** for implementing coordination
  - tuple space is a **distributed shared memory** coupled with certain operations
  - tuple space originated in the Linda coordination language created in the mid-1980's by David Gelernter
    - further developed and refined over the years by researchers
  - commercial implementations of the tuple space **model**:
    - *C-Linda* from Scientific Computing Associates, Inc.
    - *TSpaces* from IBM
    - *JavaSpaces* from Sun
  - many free software implementations
    - simplistic, incomplete, non-scalable, research-oriented, etc.
  - Ark contains a tuple space implemented from scratch and tailored for a measurement infrastructure
    - hope to release under GPL as a standalone piece of software

Tuple Space

- before proceeding,
  it's worth noting what a tuple space is ***not*** ...

Tuple Space

- before proceeding,
  it's worth noting what a tuple space is ***not*** ...

- **not** MPI, OpenMP, etc.
- **not** a distributed hash table (DHT)
- **not** a distributed database
- **not** a new routing system/protocol for IP packets
  - not a new BGP or IGP; not a new overlay (RON, GENI)
- **not** a new DNS

Tuple Space

- before proceeding,
  it's worth noting what a tuple space is ***not*** ...

- **not** for coding measurement logic
  - tuple space is a ***medium*** not an implementation language
  - only for coordinating measurement **activity**
    - write measurement tools in Ruby, C/C++, etc.
    - hook up measurement tool to the tuple space
    - use tuple space to control, direct, and glue together measurement tools
- **not** for transferring bulk data
  - transfer only coordination (command/control) data/metadata
  - transfer bulk data with a separate TCP, FTP, HTTP, SCP, etc. connection

What ***is*** a tuple space, then?

Tuple Space

- implementation-wise, a tuple space is closest in concept to a database
  - similar client-server design
  - e.g., global tuple space is a datastore hosted by a server process running at CAIDA
- superficial resemblance to publish/subscribe systems

Tuple Space

- tuple space contains *tuples*
  - multiset: can have any number of tuples with the same value
- tuples are an **ordered** collection of values of possibly mixed type (int, float, string, ...)
  - can have any number of components
  - up to users to define meaning of tuples
    - meaning rests solely on implicit convention
    - advantage: no formal (database-like) schema required or declared

Tuple Space

- tuple space contains *tuples*
  - multiset: can have any number of tuples with the same value
- tuples are an **ordered** collection of values of possibly mixed type (int, float, string, ...)
  - can have any number of components
  - up to users to define meaning of tuples
    - meaning rests solely on implicit convention
    - advantage: no formal (database-like) schema required or declared
- examples:
  - `("composer", "Bach", 1685, 1750)`
  - `("Bach", 1011, "Cello Suite No. 5 in C minor")`
  - `("J.A. Bach", "J.S. Bach")`
  - `("J.S. Bach", "C.P.E. Bach")`
  - `("J.S. Bach", "W.F. Bach")`

Tuple Space

- tuple space is an associative memory
  - *match* user-supplied *template* against all tuples
  - template is like a tuple except it can have wildcards (*)
    - `(("J.S. Bach", "C.P.E. Bach"))`
    - `(("J.S. Bach", *))`
  - template *matches* tuple if
  - template and tuple have same number of components, **and**
  - values at corresponding positions in template and tuple *match*:
    - literal value only *matches* the same value
    - wildcard always *matches* any value of any type

Tuple Space

- tuple space is an associative memory
  - *match* user-supplied *template* against all tuples
  - template is like a tuple except it can have wildcards (*)
    - `(("J.S. Bach", "C.P.E. Bach"))`
    - `(("J.S. Bach", *))`
  - template *matches* tuple if
    - template and tuple have same number of components, **and**
    - values at corresponding positions in template and tuple *match*:
      - literal value only *matches* the same value
      - wildcard always *matches* any value of any type
- examples of template matching:
  - `(("J.S. Bach", *))` *matches* `("J.S. Bach", "C.P.E. Bach")`
  - `(("J.S. Bach", *))` *does* **not** *match* `("J.S. Bach", 1685, 1750)`
  - `(("J.S. Bach", *, *))` *matches* `("J.S. Bach", 1685, 1750)`
  - `((*, 1685, *))` *matches* `("J.S. Bach", 1685, 1750)`

Tuple Space

- 3 fundamental tuple space operations:
  - **write**(*tuple*)
    - adds a tuple
  - **read**(*template*)
    - returns a copy of a matching tuple (tuple remains in tuple space)
    - blocks until a matching tuple is added to the tuple space
  - **take**(*template*)
    - removes matching tuple from tuple space and returns it
    - blocks until a matching tuple is added to the tuple space

Tuple Space

- properties beneficial for coordination:
  - designed explicitly for concurrency
    - burden of locking shared space on system, **not** on user
    - automatic mutual exclusion: system guarantees that only one process can remove a given tuple with *take* operation
  - operations block waiting for matching tuple
    - supports **decoupling in *time***
    - reader and writer processes may have different or non-overlapping lifetimes
- tuples are not addressed to an explicit recipient
  - supports **decoupling in *space***
  - reader and writer processes don't need to know the identity or location or even existence of each other
  - allows dynamically changing, open-ended set of participants

Coordination Patterns

- **semaphores**
  - enforce mutual exclusion in resource access or use
  - tuple == semaphore
  - library book metaphor:
    - book on shelves => available => semaphore free
    - book missing => not available => semaphore locked

Coordination Patterns

- **semaphores**
  - enforce mutual exclusion in resource access or use
  - tuple == semaphore
  - library book metaphor:
    - book on shelves => available => semaphore free
    - book missing => not available => semaphore locked
  - e.g., to prevent concurrent probing into a given AS:
    - *setup*: `write("AS701")`
    - *client 1*: `take("AS701"); doit(); write("AS701")`
    - *client 2*: `take("AS701"); doit(); write("AS701")`
  - set allowed level of parallelism or concurrent access by initializing with multiple tuples:
    - *setup*: `write("AS701"); write("AS701")`
    - *client 1*: `take("AS701"); doit(); write("AS701")`
    - *client 2*: `take("AS701"); doit(); write("AS701")`

Coordination Patterns

- **barrier synchronization**
  - block fast-running tasks until all tasks reach a certain point in processing or execution, after which all tasks become unblocked
    - e.g., want all measurement tasks to start at same time at beginning of each stage of a multistage measurement

Coordination Patterns

- **barrier synchronization**
  - block fast-running tasks until all tasks reach a certain point in processing or execution, after which all tasks become unblocked
    - e.g., want all measurement tasks to start at same time at beginning of each stage of a multistage measurement
  - one implementation approach: for 3 processes, *A*, *B*, & *C*:
    - *A*: `write("A-done"); read("B-done"); read("C-done")`
    - *B*: `write("B-done"); read("A-done"); read("C-done")`
    - *C*: `write("C-done"); read("A-done"); read("B-done")`
  - another approach: for general *n* processes--use counter:
    - *global setup*: `write("working", n);`
    - *each process*:

```
wait_for_all() {
    (x, n) = take("working", *);
    write("working", n-1);
    read("working", 0);
}
```

Coordination Patterns

- **distributed data structures**
  - lists, queues, trees, graphs, ... can be built with tuples
  - data structures exist on their own independently of processes
  - processes concurrently manipulate these data structures
  - provides a foundation for distributed processing and problem solving

Coordination Patterns

- **distributed data structures**
  - lists, queues, trees, graphs, ... can be built with tuples
  - data structures exist on their own independently of processes
  - processes concurrently manipulate these data structures
  - provides a foundation for distributed processing and problem solving
  - e.g., can implement producer-consumer pattern supporting arbitrary number of consumers and producers:

data structure: `(1, “Bach”);(2, “Mozart”);(“head”, 1);(“tail”, 2)`

```
produce(val) {
  (x, n) = take(“tail”, *);
  write(“tail”, n+1);
  write(n, val);
}
```

```
consume() {
  (x, n) = take(“head”, *);
  write(“head”, n+1);
  (y, val) = take(n, *);
  return val;
}
```

Coordination Patterns

- **Bag-of-Tasks (aka Master-Worker) scheduling**
  - decompose complex or repetitive jobs and parcel out pieces to workers
  - automatic distribution: no central authority that assigns work
  - automatic load balancing: each worker runs at its own pace and a slow worker doesn't cause faster workers to idle

Coordination Patterns

- **Bag-of-Tasks (aka Master-Worker) scheduling**
  - decompose complex or repetitive jobs and parcel out pieces to workers
  - automatic distribution: no central authority that assigns work
  - automatic load balancing: each worker runs at its own pace and a slow worker doesn't cause faster workers to idle
  - e.g., want to probe every routed /24, balancing load across team of 30 machines

data structure: `("task", "192.168.0.0/24")`

```
master(tasks) {
  for t in tasks {
    write("task", t);
  }
}
```

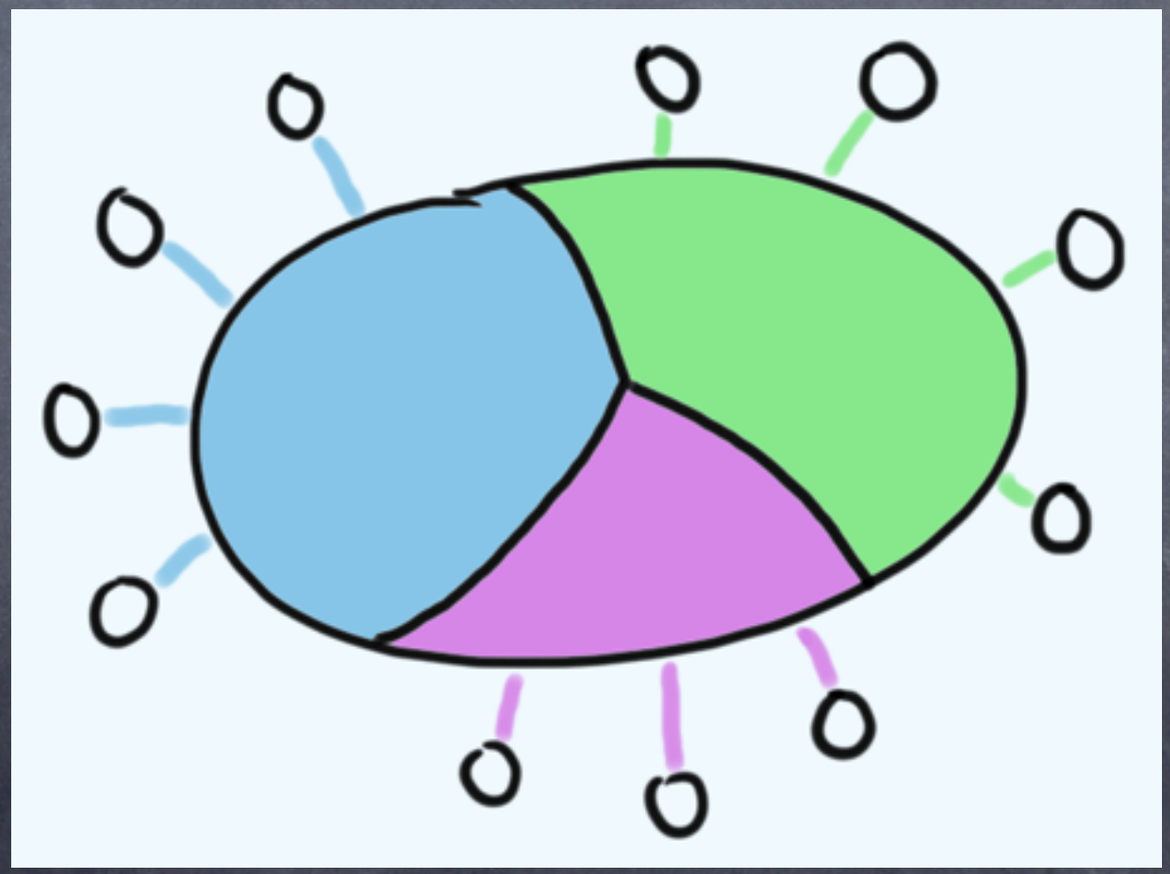
```
worker() {
  forever {
    (x, t) = take("task", *);
    doit(t);
  }
}
```

Tuple Space Features

- tuple space **implementation** in Ark is far more sophisticated than basic **model** described so far
- full list of features:
  - multiple tuple space regions
  - local & global scopes
  - private one-to-one and group communication
  - fine-grained per-region privileges
  - many operations: non-blocking variants, iteration, ...

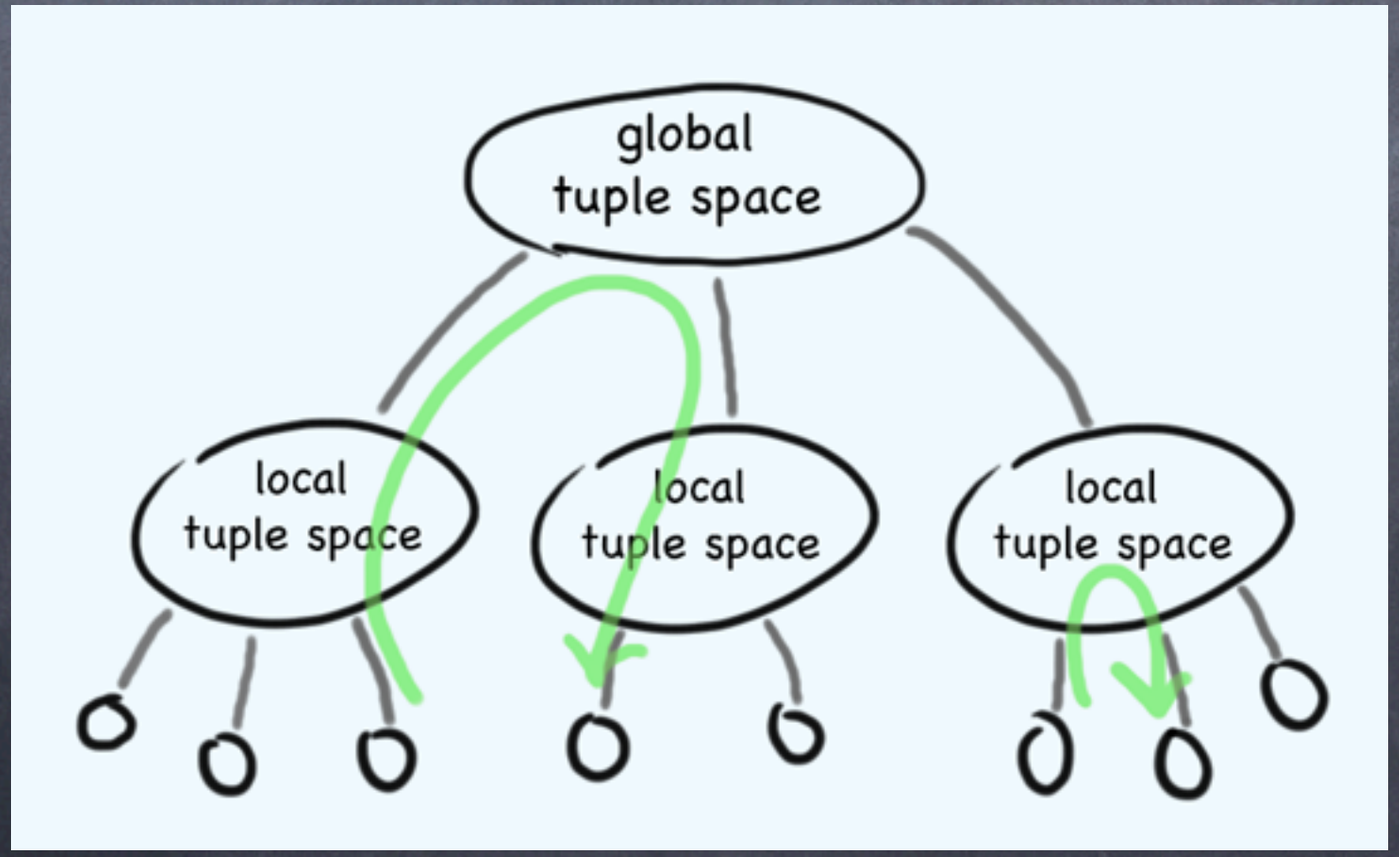
Tuple Space Features

- multiple disjoint tuple space *regions*
  - partition communication space for privacy and to prevent interference (cross talk)



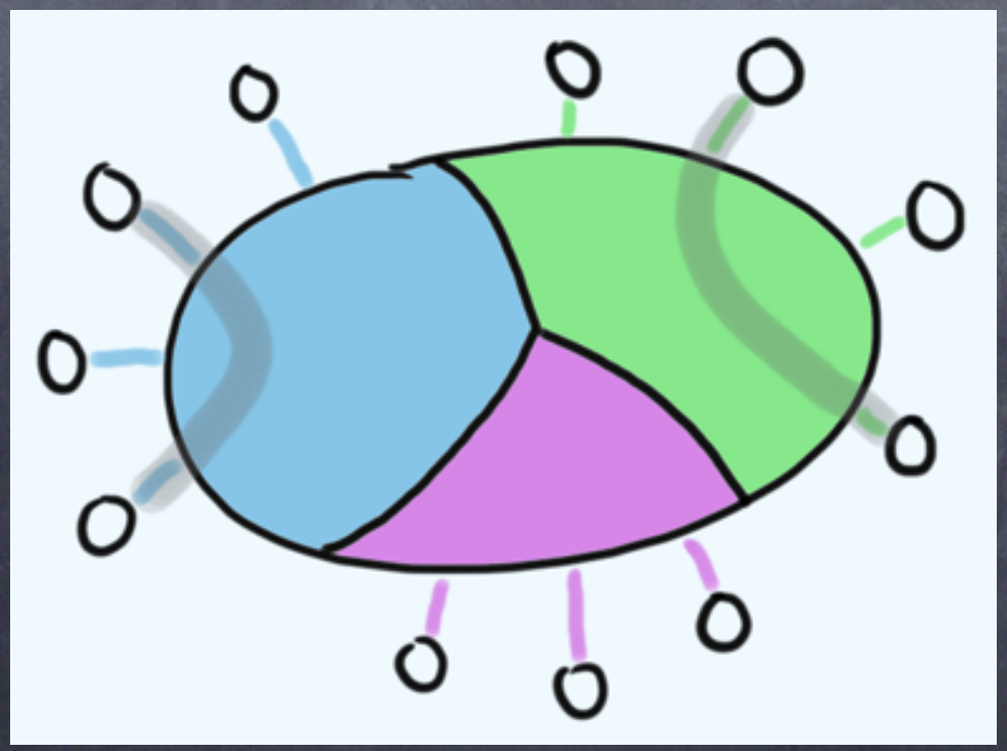
Tuple Space Features

- two scopes:
  - **local**: tuple space regions local to given node
    - only processes on node can access regions
  - **global**: tuple space regions at central server, outside all nodes
    - processes from all nodes can access regions
    - all **inter**-node communication happens in global regions; no direct node-to-node communications allowed



Tuple Space Features

- communication patterns:
  - (private) **one-to-one** communication
  - **group** communication
    - that is, many-to-many communication by subset of processes
    - group communication implemented with regions
      - having access to multiple regions = “belonging to multiple groups”
  - **all-to-all** communication
    - special case of group communication
    - all processes have access to local and global *commons* regions



Tuple Space Features

- can **pass file descriptors** over local tuple space
  - for gaining access to ...
    - open files
    - services (accessed with sockets)
    - tuple space regions (via sockets)
  - for granting selective access to resources to sandboxed measurement processes

Tuple Space Features

- operations:
  - **write**(*tuple*)
  - **read**(*template*); **take**(*template*)
  - **readp**(*template*); **takep**(*template*)
    - non-blocking versions of **read** and **take**
    - if a matching tuple currently exists in tuple space, then return it; else return nil
  - **read_all**(*template*)
    - returns all existing tuples that match template
  - **monitor**(*template*)
    - returns all existing tuples that match template, **and** returns all future tuples that match

Tuple Space Features

- operations (continued):
  - *p* = **remember_peer**(); **forget_peer**(*p*);
  - **write_to**(*p*, *tuple*); **reply**(*tuple*)
    - send private one-to-one communication
  - **take_priv**(*template*); **takep_priv**(*template*)
    - receive private one-to-one communication
- **forward_to**(*p*, *tuple*)
  - send private one-to-one communication with masquerading of sender
- **pass_access_to**(*p*, *file_descriptor*, *tuple*)
  - pass arbitrary open file descriptor to another local process
  - pass access to tuple space region to another local process
    - one mechanism for granting group membership

Tuple Space Features

- fine-grained per-region privileges:
  - can **read** tuples
  - can **write** tuples
  - can **write sticky** tuples
  - can **take** tuples
  - can **forward** tuples
  - can **pass access rights** (file descriptors)

Examples

- 4 examples of using Ark's tuple space in practice
  - real source code in Ruby
  - need only a few lines of initialization and the code for measurement logic to run
    - in particular, not hand waving away critical details or complexity
- in sample code, ***ts*** is a connection to a tuple space region

Example 1

- simple **ping service**:
  - client supplies address to ping
  - server performs ping and returns RTT

Example 1: ping service

client:

```
req_id = ts.gen_id()    # globally unique ID
ts.write ["PINGv1", req_id, "192.168.0.5"]
result = ts.take ["PINGv1-RESULT", req_id, nil]
puts "RTT = " + result[2]
```

---

server:

```
loop do
   request = ts.take ["PINGv1", nil, nil]
   req_id, addr = request[1..2]
   rtt = ping(addr)
   ts.write ["PINGv1-RESULT", req_id, rtt]
end
```

Example 1: ping service

- beneficial properties:
  - ***space decoupling:*** client does not need to know *who* or *where* the server is
    - client only needs to know the *request protocol* to use the ping service
    - server can be moved around without affecting clients

Example 1: ping service

- beneficial properties:
  - ***space decoupling:*** client does not need to know *who* or *where* the server is
    - client only needs to know the *request protocol* to use the ping service
    - server can be moved around without affecting clients
  - ***time decoupling:*** client does not need to wait for the server to be running before making its request
    - shields clients from planned or unexpected server shutdowns, restarts, and location migration; for example:
      1. server dies, or is shut down and moved
      2. client makes request, and blocks on *take*
      3. server starts up, and performs request
      4. client receives result, none the wiser

Example 1: ping service

- beneficial properties (cont'd):
  - ***automatic load balancing:*** simply launch multiple server processes with the **same** code
    - fast servers will automatically service more requests than slower servers
    - server instances can be started up on different hosts or at different locations
    - no need to make any configuration changes to activate load balancing across machines or sites

Example 1: ping service

- beneficial properties (cont'd):
  - ***automatic load balancing:*** simply launch multiple server processes with the **same** code
    - fast servers will automatically service more requests than slower servers
    - server instances can be started up on different hosts or at different locations
    - no need to make any configuration changes to activate load balancing across machines or sites
  - ***request-result decoupling:*** the client making the request need not be same client that processes the result
    - can have one client issuing requests (based on, say, user interaction); another client can process, analyze, archive, or visualize the result
    - analysis client can be a different thread, or a different process altogether on a different host at a different location

Example 2

- **geography-aware** ping service:
  - client supplies **server geographic location** and address to ping
  - server performs ping and returns RTT

Example 2: ping service (geo)

client:

```
req_id = ts.gen_id()
ts.write ["PINGv2", req_id, "Europe", "192.168.0.5"]
result = ts.take ["PINGv2-RESULT", req_id, nil]
puts "RTT = " + result[2]
```

---

server in Europe:

```
loop do
   request = ts.take ["PINGv2", nil, "Europe", nil]
   req_id, addr = request.values_at(1, 3)
   rtt = ping(addr)
   ts.write ["PINGv2-RESULT", req_id, rtt]
end
```

Example 2: ping service (geo)

server in Asia:

```
loop do
   request = ts.take ["PINGv2", nil, "Asia", nil]
   req_id, addr = request.values_at(1, 3)
   rtt = ping(addr)
   ts.write ["PINGv2-RESULT", req_id, rtt]
end
```

Example 2: ping service (geo)

client:

```
req_id = ts.gen_id()
ts.write ["PINGv2", req_id, nil, "192.168.0.5"]
result = ts.take ["PINGv2-RESULT", req_id, nil]
puts "RTT = " + result[2]
```

- client can leave geo parameter unspecified to allow any server to fulfill request
  - no change required in server code
  - feature comes for free from the tuple matching algorithm
    - normally, wildcards appear in the template, but they can also appear in the tuple, acting like “don’t cares”
    - called *inverse structured matching*

Example 3

- **geography- and AS-aware** ping service:
  - client supplies address to ping and two optional parameters:
    - **server geographic location**
    - **server AS number**
  - server performs ping and returns RTT

Example 3: ping service (geo+AS)

client:

```
req_id = ts.gen_id()
ts.write ["PINGv3", req_id, "Europe", "AS3333",
          "192.168.0.5"]
result = ts.take ["PINGv3-RESULT", req_id, nil]
puts "RTT = " + result[2]
```

---

client: `ts.write ["PINGv3", req_id, "Europe", nil, "192.168.0.5"]`

---

client: `ts.write ["PINGv3", req_id, nil, "AS3333", "192.168.0.5"]`

---

client: `ts.write ["PINGv3", req_id, nil, nil, "192.168.0.5"]`

Example 3: ping service (geo+AS)

- This server can handle all of the previous client requests ....

---

server in Europe in AS3333:

```
loop do
   request = ts.take ["PINGv3", nil, "Europe",
                      "AS3333", nil]
   req_id, addr = request.values_at(1, 3)
   rtt = ping(addr)
   ts.write ["PINGv3-RESULT", req_id, rtt]
end
```

Example 3: ping service (geo+AS)

- This server can handle all of the previous client requests, **but what if the client requests AS776?**

```
ts.write ["PINGv3", req_id, "Europe", "AS776",
          "192.168.0.5"]
```

We'd like the **same** server to handle all requests for unknown ASes in Europe. How?

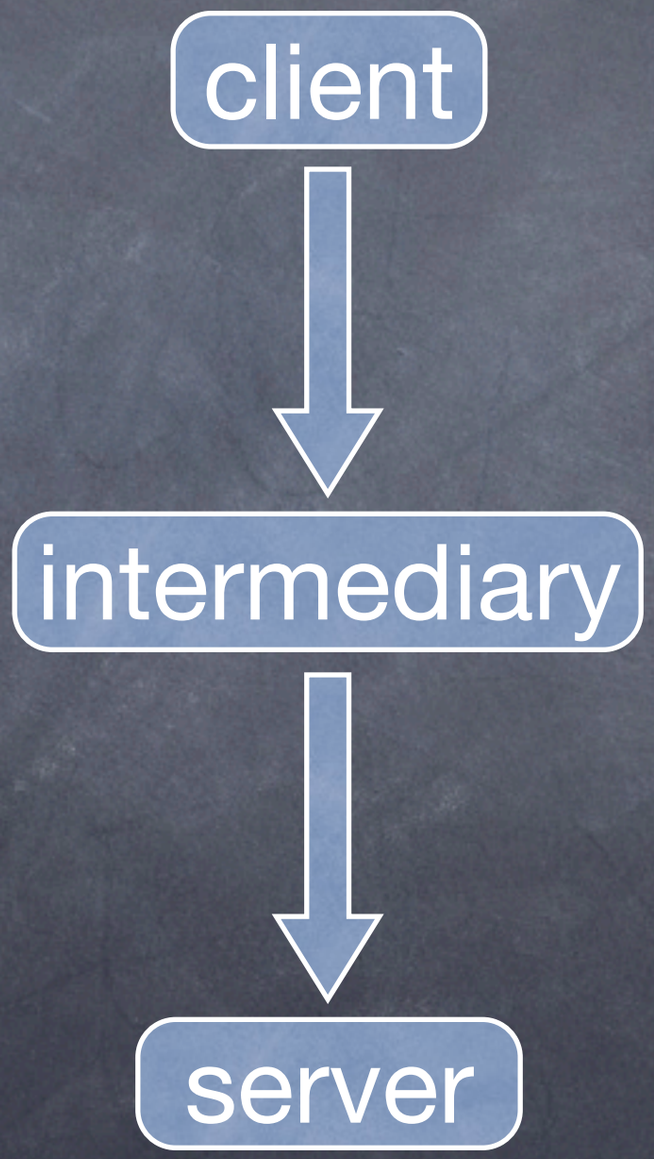
---

server in Europe in AS3333:

```
loop do
   request = ts.take ["PINGv3", nil, "Europe",
                      "AS3333", nil]
   req_id, addr = request.values_at(1, 3)
   rtt = ping(addr)
   ts.write ["PINGv3-RESULT", req_id, rtt]
end
```

Example 3: ping service (geo+AS)

- in this case, tuple matching is not powerful enough
- need *service interposition:*

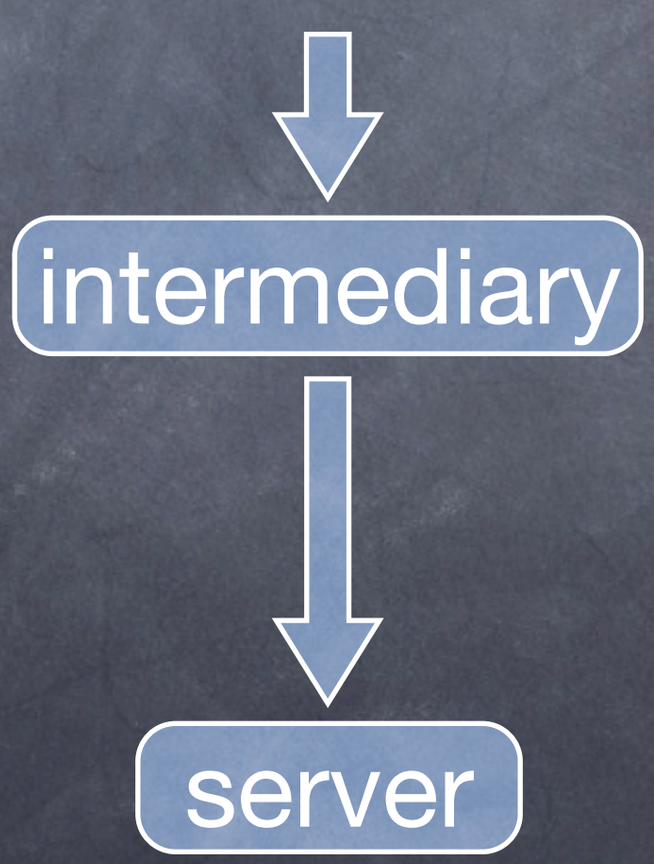


Example 3: ping service (geo+AS)

- in this case, tuple matching is not powerful enough
- need *service interposition:*

client

```
["PINGv3", req_id, "Europe", "AS776", "192.168.0.5"]
```



Example 3: ping service (geo+AS)

- in this case, tuple matching is not powerful enough
- need *service interposition:*

client

```
["PINGv3", req_id, "Europe", "AS776", "192.168.0.5"]
```

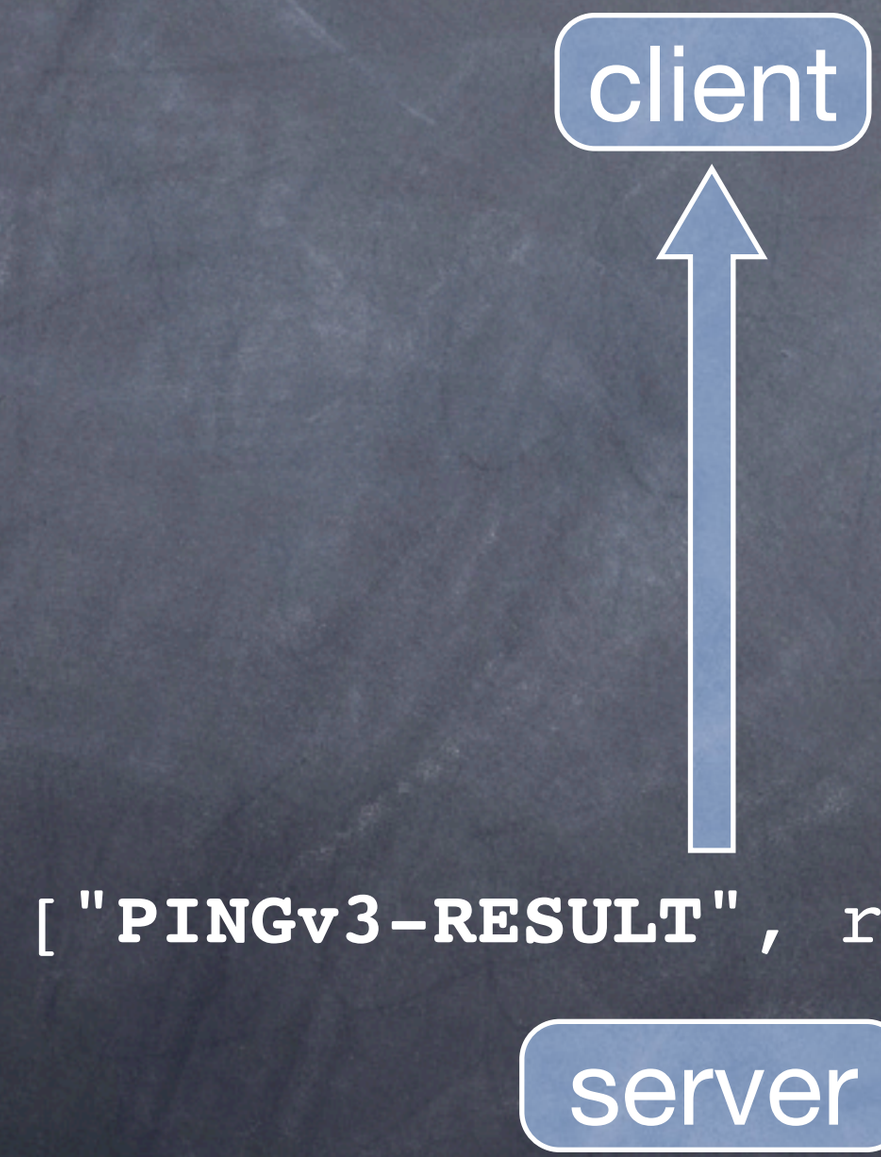
intermediary

```
["PINGv3-REQ", req_id, "Europe", nil, "192.168.0.5"]
```

server

Example 3: ping service (geo+AS)

- in this case, tuple matching is not powerful enough
- need *service interposition:*



Example 3: ping service (geo+AS)

client: *unchanged*

---

revised server:

```
loop do
   request = ts.take ["PINGv3-REQ", nil, "Europe",
                      "AS3333", nil]
   req_id, addr = request.values_at(1, 3)
   rtt = ping(addr)
   ts.write ["PINGv3-RESULT", req_id, rtt]
end
```

Example 3: ping service (geo+AS)

intermediary:

```
loop do
   request = ts.take ["PINGv3", req_id, nil, nil, nil]
   req_id, addr = request.values_at(1, 4)
   geo, asnum = find_server(request)
   ts.write ["PINGv3-REQ", req_id, geo, asnum, addr]
end
```

---

revised server:

```
loop do
   request = ts.take ["PINGv3-REQ", nil, "Europe",
                      "AS3333", nil]
   req_id, addr = request.values_at(1, 3)
   rtt = ping(addr)
   ts.write ["PINGv3-RESULT", req_id, rtt]
end
```

Example 3: ping service (geo+AS)

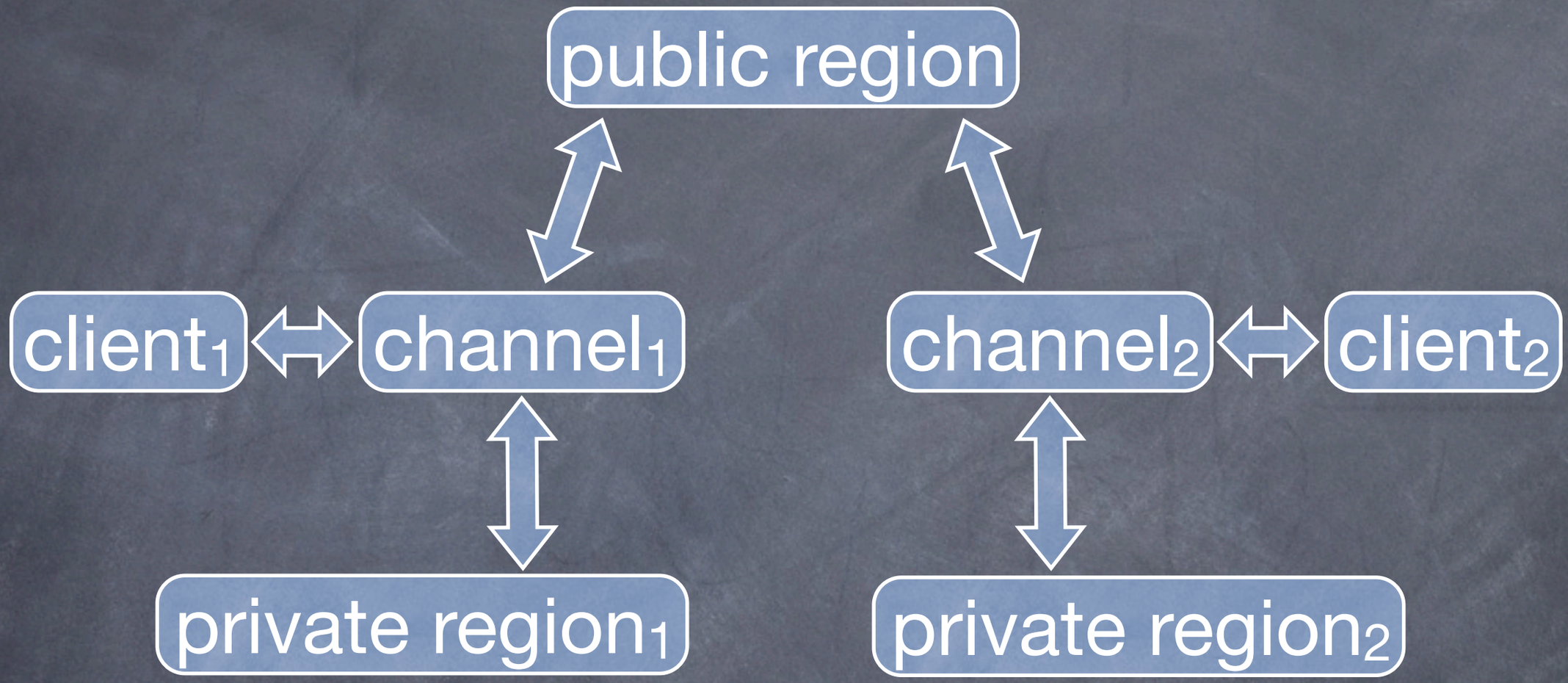
- service interposition is a general technique useful for:
  - providing a simplified interface to complex services
  - implementing complex or expensive request dispatch
  - translating / bridging disparate protocols

- supporting legacy protocols
  - server only implements new protocol
  - intermediary translates older protocols
  - remove intermediary after all clients have been upgraded

Example 4

- ping service with **private communication**:
  - client supplies address to ping
  - server performs ping and returns RTT using **private 1-to-1 communication** with client

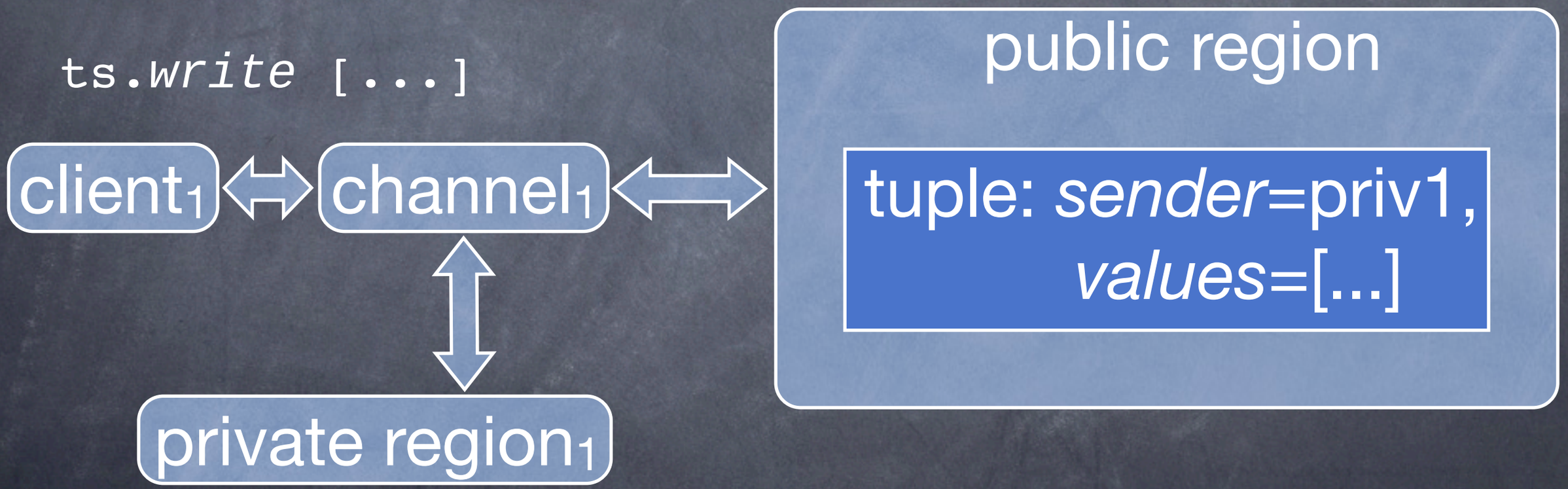
Example 4: ping service (private)



- each client is always connected to exactly two regions:
  - a **public region** that might be shared with other clients
  - a **private region** to which only the client has access
    - impossible to share a private region even if the client wishes to

Example 4: ping service (private)

- a client is identified by its private region
- each tuple contains a ***sender*** field which records the private region of the client that wrote the tuple



Example 4: ping service (private)

client:

```
ts.write ["PINGv4", "192.168.0.5"]
result = ts.take_priv ["PINGv4-RESULT", nil]
puts "RTT = " + result[1]
```

take_priv => take tuple from per-client private region

---

server:

```
loop do
   request = ts.take ["PINGv4", nil]
   addr = request[1]
   rtt = ping(addr)
   ts.reply ["PINGv4-RESULT", rtt]
end
```

reply => insert tuple into private region of client that sent last retrieved tuple

Example 4: ping service (private)

- can remember sender to engage in dialogue:

```
tuple = ts.take [...]
peer = ts.remember_peer()
ts.write_to(peer, [...])
ts.take_priv [...]
ts.write_to(peer, [...])
ts.take_priv [...]
```

- can pass file descriptor:

```
file = File.open(“abc”)
ts.pass_access_to(peer, file, [“FILE”, “abc”])
```

Example 4: ping service (private)

- private regions provide a concurrency model similar to Erlang's asynchronous message passing
  - client can call ***read_priv*** or ***take_priv*** with a template to pull out tuples in whatever order it wishes
  - client can use a template of **[ ]** to pull out the first tuple

Status

- implemented:
  - tuple space in Ruby
    - also have (very) incomplete tuple space implementation in Erlang
  - Ruby scripts to perform scamper-based global measurements
  - data collection

- unimplemented:
  - security subsystem
    - not ready for use by non-trusted users
  - subsystem for deploying arbitrary measurement software
    - for now, manually deploy software
  - scalable tuple matching algorithm
    - very hard multikey matching problem
  - handling failure of machine hosting global tuple space
    - need logging of tuple operations, and replaying on restart

Conclusions

- ***coordination*** is a common need in a measurement infrastructure
  - hence, worth supporting directly
- Ark provides a ***tuple space*** for implementing coordination
  - simple to use, and powerful enough for typical situations
  - enables a highly decoupled system that is flexible and adapts to change
    - e.g., handles addition, removal, or movement of clients and servers
  - allows anyone to create and run servers
  - allows users/researchers to build upon each other's work

Thanks!

ark-info@caida.org