The **Archipelago** *Measurement Infrastructure*

Young Hyun
CAIDA

7th CAIDA-WIDE Workshop, Nov 2006

Outline

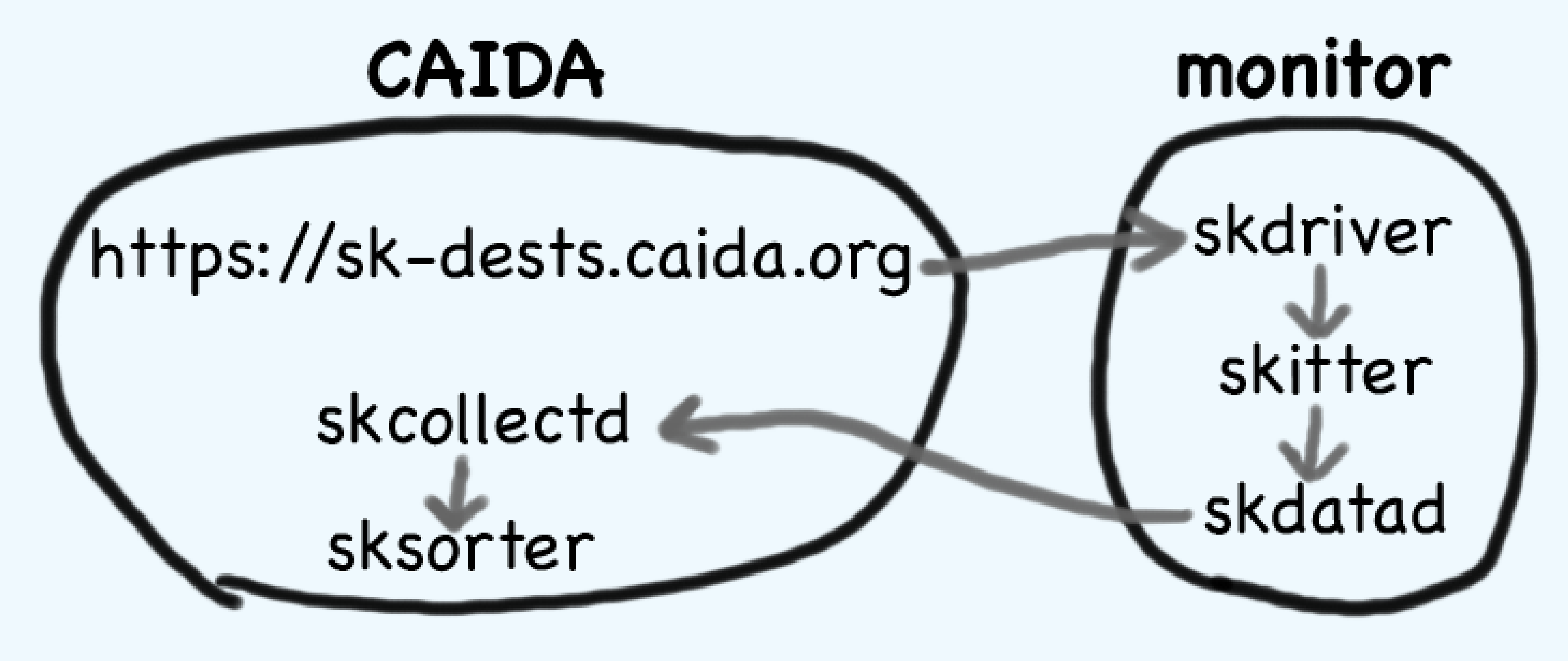
- background
- goals
- architecture
- status

Background

- CAIDA’s Macroscopic Topology Project
  - represents our main effort in **active** measurement
  - more than 8 years of data collection
  - running skitter on 20-25 monitors worldwide
  - > 12 billion complete skitter traces (as of Nov 2006)
  - CAIDA has used data for
    - AS graph poster
    - AS ranking
    - Internet Topology Data Kit (ITDK)
    - various topology analyses

Background

- terminology
  - skitter *tool*
    - performs parallel traceroutes
  - skitter *infrastructure*
    - distributes destination lists to monitors, performs measurements, and collects traces
    - skitter tool + other software + web server



Introduction

- Archipelago (Ark) is CAIDA's next generation active measurement infrastructure
  - software + hardware (machines)
- replaces skitter infrastructure
  - *skitter infrastructure* = currently deployed software = **means**
- Ark is an upgrade to the **means** of the Macroscopic Topology Project
- the Project will go on, and skitter-like measurements will be main focus of Ark

Introduction

- Ark will have **minimal** impact on researchers currently **using** skitter data
  - same type of data (just in different file format)
  - same type of global, large-scale traceroute measurements
- Ark will have greater impact on researchers wanting **to do** active measurement
  - allows sophisticated, dynamic, etc. destination lists for skitter-like measurements
    - better employ available resources to get more bang for buck
  - beyond traceroute measurements

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

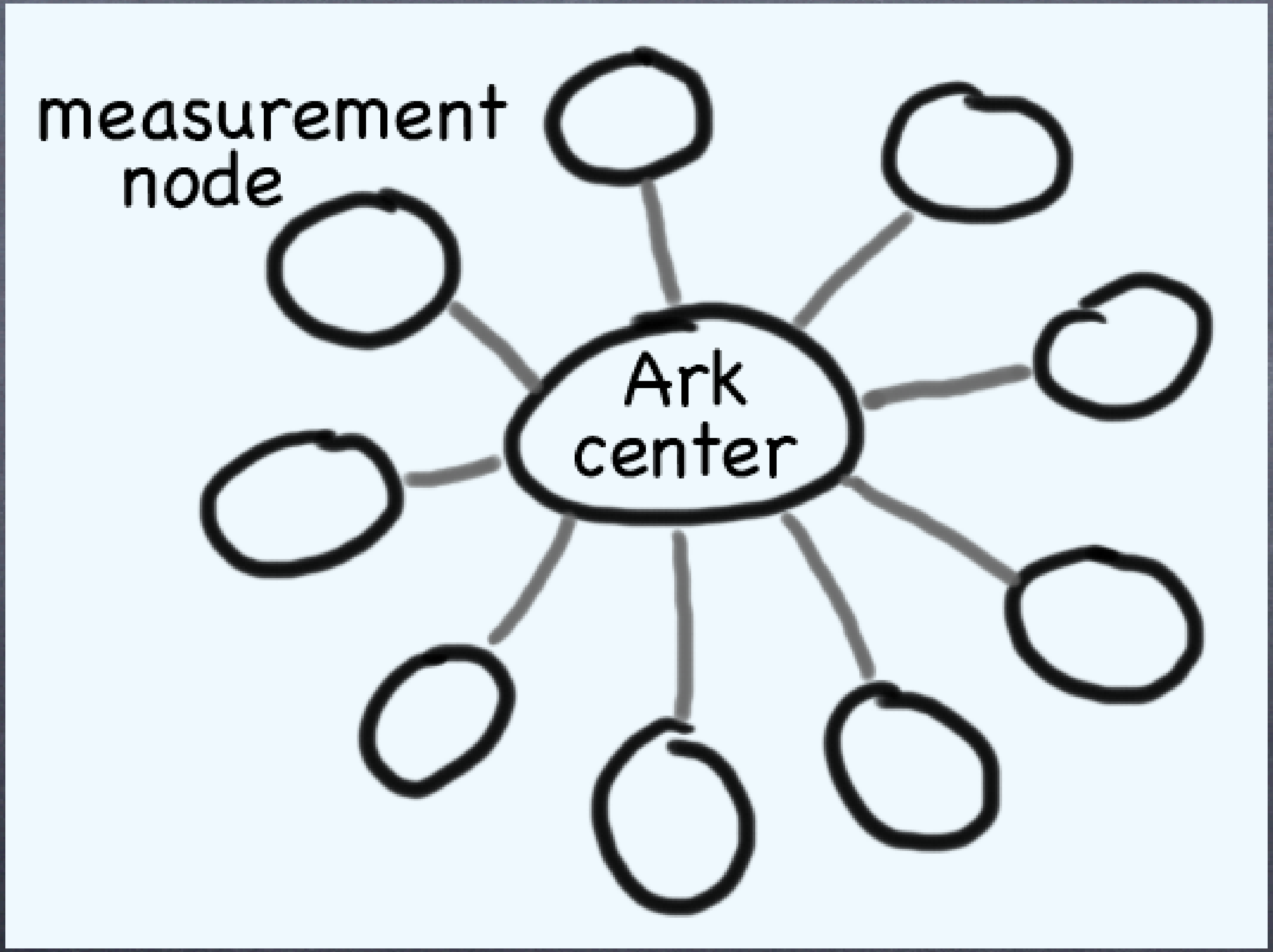
- greater scalability and flexibility
  - scalability in system management, monitor deployment, measurement efficiency, resource utilization
  - flexibility in measurement method, scheduling, data collection
- platform for measurement tool development, experimentation, deployment
  - raise level of abstraction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

Architecture

- topology
- security
- communication & coordination
- software installation & execution
- data storage & management

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 (same as skitter)



Architecture

- topology
- **security**
- communication & coordination
- software installation & execution
- data storage & management

Security Features

- multiple levels of trust:
    - **stranger** (general public) -- no trust
    - **acquaintance** -- some trust
    - **collaborator** -- medium to high trust
- secure communication
- process isolation (sandboxing)
- rate & resource limiting
- packet filtering
- fine-grained access control of resources

Security Analysis

- *"it is secure"* has no meaning without context
- **secure against what?**
  - *who*, *what*, and *where* are the threats?
  - how do you mitigate each particular threat?

Security Analysis

- threat from **3rd party**: eavesdropping & taking control
- mitigation:
  - all communication over SSL
    - custom root certificate; check client & server certificates
  - small, well-defined set of open server ports
    - base operation: only SSH--all other connections opened out from node
    - ports of measurement tools; e.g., server-side of bandwidth estimation tool
  - closed membership
    - attacker is outsider: only machines of collaborators may join system
    - contrast with open systems where first line of attack is to join system
  - communication in star topology
    - nodes must directly trust only the central server
    - no $O(n^2)$ node-to-node authentication that can be subverted

Security Analysis

- threat from **public user**: privilege escalation & launching attacks
- mitigation:
  - execute in sandbox
    - FreeBSD jail: even root access doesn't compromise system
  - restricted measurement capabilities
    - traceroute- and ping-like measurements only
    - no TCP connections; no UDP packets (not even DNS)
  - rate limiting; packet filtering by destination address
  - no ability to read/write local files
    - not even as root--system immutable flag

Security Analysis

- threat from **collaborator**: privilege escalation & denial of service (DoS) of Ark itself
- mitigation:
  - enforce levels of confinement: completely open to restricted
  - optional sandbox (FreeBSD jail)
  - optional rate limiting & packet filtering
  - fine-grained access control of files & privileged resources (e.g., raw sockets)
  - filesystem resource limits
  - FreeBSD jail-based CPU & memory resource limits
  - partitioning of communication space for privacy and to prevent interference
  - full protection against DoS not possible
    - concerned more about accidental DoS than intentional

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

- rejected approach: access control lists (ACL)
  - ACL is a list of (user, rights) pairs attached to object
    - e.g., [(Alice, read/write), (Bob, read)] for file /data/stuff.txt
  - authorization: look up **identity** of principal in ACL, and grant enumerated rights
- drawbacks:
  - requires authentication to establish identity
  - identity must be established across machines
  - ACLs must be kept up-to-date across machines and in the face of network failure or partitioning
  - potential for inconsistent or incomplete ACLs
    - that is, hard to correctly implement policy across machines
  - hard to delegate authorization duties
  - hard to pass along access rights to others

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

Capabilities

- advantages:
    - no authentication required (no identity checks)
    - no need to establish identities
    - no ACL-like metadata that must be kept up-to-date
    - no possibility for inconsistency or incompleteness since no metadata exists
    - can delegate authorization duties by granting *authorization capability*
    - can selectively grant rights to others
    - can enforce Principle of Least Privilege

Capabilities

- potential drawbacks and difficulties:
  - hard to track exactly who used a resource
  - hard to enumerate all principals who can potentially access a resource
  - hard to revoke capabilities on per-principal basis
  - *confinement problem*--hard to control willful propagation of capabilities
    - not compromise of system, just Alice intentionally giving (sharing) a capability to Bob
- these issues **may or may not**
  - exist in a given implementation of capabilities
  - matter for a given use of capabilities

Capabilities

- real-world examples of capability-like objects:
  - car keys
    - car doesn't check your identity before starting engine
    - can give car keys to valet without worrying about valet entering your house
  - stickers for hybrid cars that permit driving in carpool lanes
    - police officer enforces carpool lane by checking for presence of sticker--simple & quick
    - police officer does **not** need to check every license plate against complete list of authorized vehicles
    - auto dealer can (theoretically) give out stickers to car purchasers
  - carnival tickets
    - tickets can be sold in multiple booths at different locations without requiring coordination or record keeping
    - ride operators simply check for possession of ticket

Capabilities

- technical example: Unix file descriptor
  - integer value refers to open file with particular rights (read/write) in kernel
    - can’t forge file descriptor
    - necessary & sufficient: I/O system calls work on file descriptor
  - pass open file descriptor from one process to another via (local) socket to grant access
  - Principle of Least Privilege
    - the process receiving an open file gains no more access than the file

Capabilities

- capabilities implementation:
  - *internal* capabilities:
    - functional object reference that can only exist within system
      - can **directly** dereference to access object
    - file descriptors for access to files, raw sockets, and tuple space regions
  - *external* capabilities:
    - non-functional object reference that can exist outside system
      - can store on disk, email to someone, etc.
      - must **indirectly** dereference to access object
    - crypto-based implementation:
      - care about **authenticity** and **integrity** of capabilities
      - similar in concept (digital signature) to X.509 certificates but for **objects and rights**, not for principals (people)
      - use *keyed-hash message authentication code* (HMAC; RFC 2104):
        - compute: MAC = *HMAC*(Object ID, Rights, Key)
        - capability is (Object ID, Rights, MAC)

Architecture

- topology
- security
- communication & coordination
- software installation & execution
- data storage & management

Communication & Coordination

- a measurement infrastructure is a distributed system with many components that must work together in complex ways toward a common goal
- ability to communicate is absolutely **necessary but not sufficient** in this environment
- 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 Facility

- coordination is usually implemented in **ad-hoc** manner on top of 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*

Tuple Space

- Ark provides a general coordination facility: *tuple space*
  - tuple space is a distributed shared memory coupled with certain operations
  - basic idea of tuple space originated in the Linda coordination language developed by David Gelernter in the 1980's
    - further developed and refined over the years by many researchers

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
  - examples of template matching:
    - (("J.S. Bach", "C.P.E. Bach")) *matches* ("J.S. Bach", "C.P.E. Bach")
    - (("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

Tuple Space Coordination Examples

- **semaphores**
  - enforce mutual exclusion in resource access or use
  - e.g., use semaphore to prevent concurrent probing into a given AS or prefix, or use multi-valued semaphores to restrict the degree of probing parallelism
    - `take(“AS701”); doit(); write(“AS701”)`
  - set allowed level of parallelism or concurrent access by varying number of “semaphore” tuples seeded in tuple space:
    - e.g., to allow two concurrent probes into AS701, prep the tuple space with `write(“AS701”); write(“AS701”)`
    - code to use semaphores remains unchanged from the case of single-valued semaphore

Tuple Space Coordination Examples

- **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”); take(“B-done”); take(“C-done”)`
    - B: `write(“B-done”); take(“A-done”); take(“C-done”)`
    - C: `write(“C-done”); take(“A-done”); take(“B-done”)`
  - another approach: for general *n* processes--use counter:

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

Tuple Space Coordination Examples

- **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;
}
```

Tuple Space Coordination Examples

- **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);
  }
}
```

Metadata in Tuple Space

- another important use: **store metadata**
  - system and node configuration
    - when node (re)starts up, it looks up its IP address in tuple space and retrieves configuration
    - supports match-making service: find node matching desired criteria (AS, prefix, performance, measurement capabilities, etc.)
  - infrastructure-wide *no-probe* list
    - records network prefixes and host addresses that, due to complaints, should not receive measurement traffic

Tuple Space Features

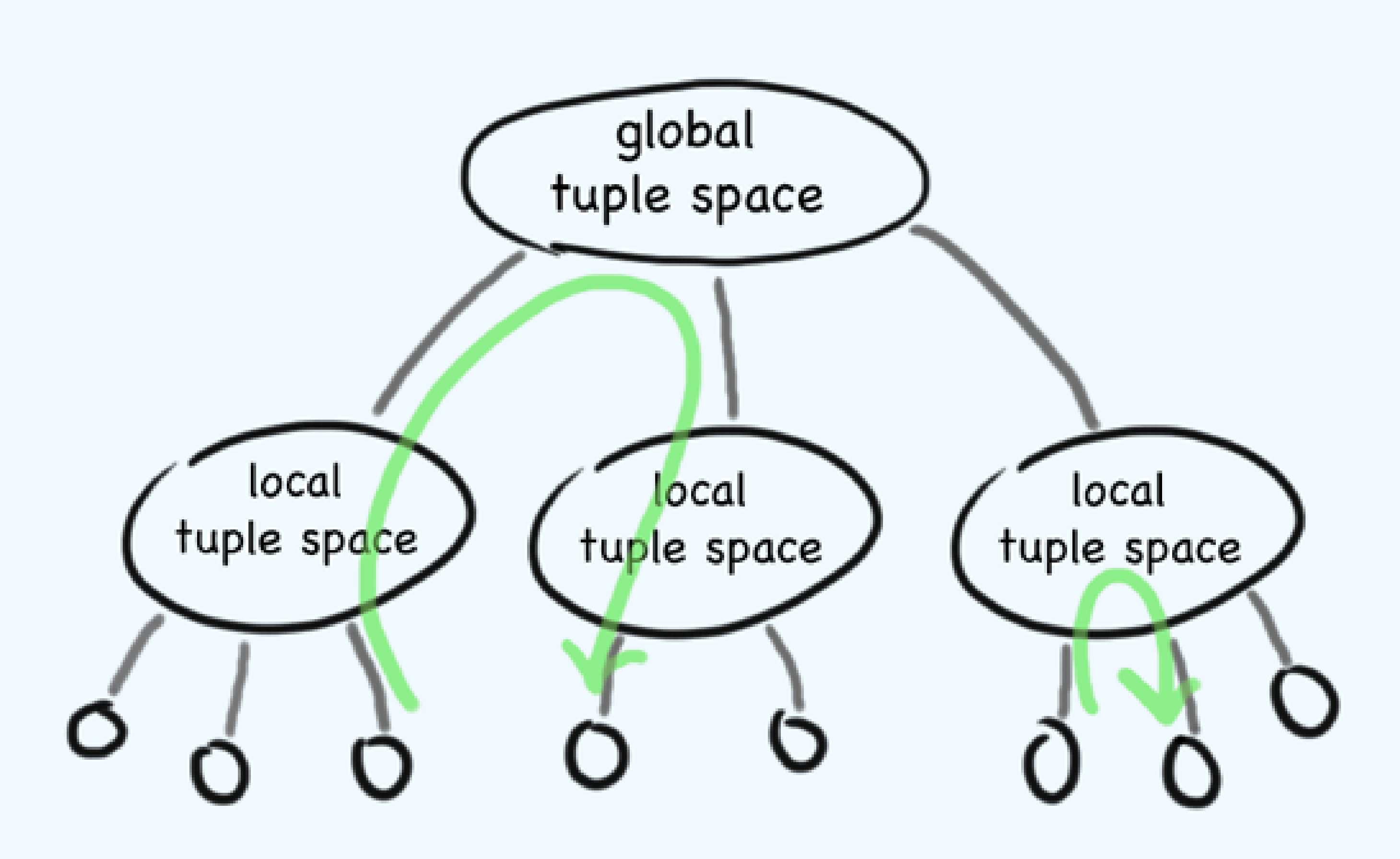
- tuple space **implementation** in Ark is far more sophisticated than basic **model** described so far
- full list of features:
  - multiple tuple space regions
  - local & global scopes
  - private one-to-one and group communication
  - tuple *qualities*
  - scalar & structured types for tuple components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

Tuple Space Features

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

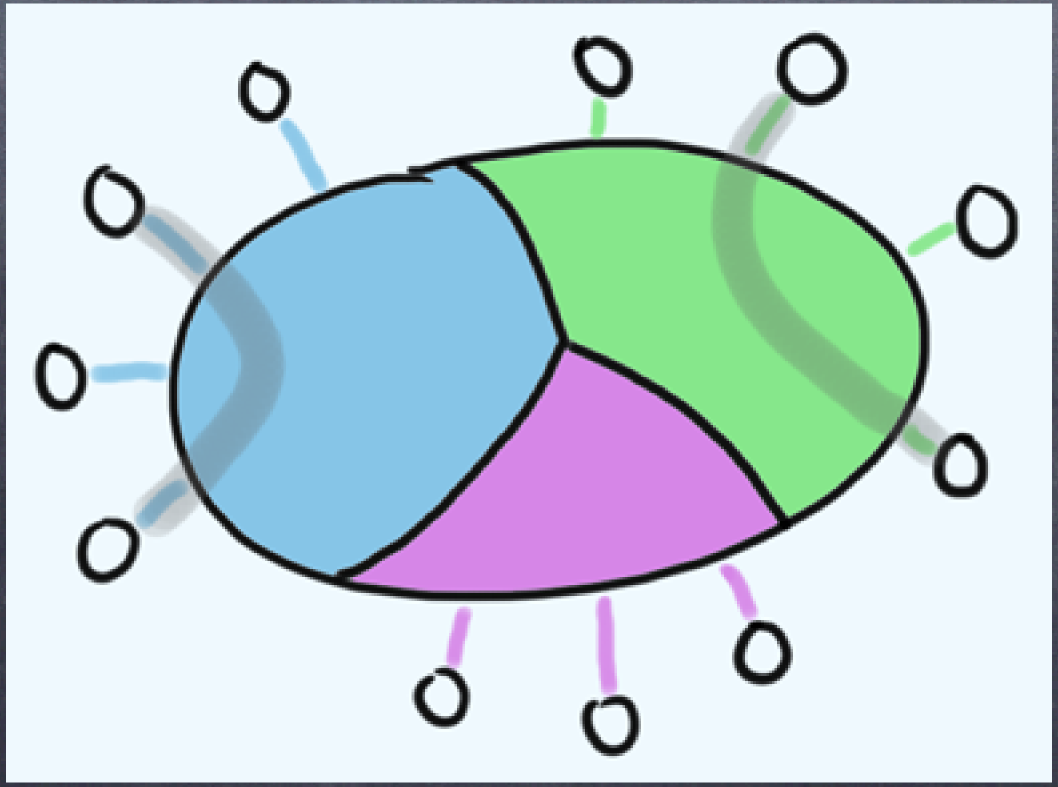
Tuple Space Features

- 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
  - private **group** communication
    - that is, many-to-many communication by subset of processes
  - public **all-to-all** communication
    - special case of group communication
  - private communication with Ark *system services*
    - special group-like communication: non-member (measurement process) communicating with a group (processes implementing a system service)



Tuple Space Features

- tuple *qualities*:
  - **sticky**
    - *sticky* tuple can only be removed (with *take*) by process that wrote it; *take* becomes *read* for all other processes
  - **precious**
    - safeguards to prevent loss of tuple following process failure
  - **auto_increment, auto_decrement**
    - more convenient use of counter tuples
- types for tuple components:
  - scalar types: **integer**, **float**, **string**
  - structured types *(experimental):* **lists & hashes**
    - *hash* as in Perl, a hash table
  - **file descriptors**
    - in local regions only

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
  - *chan* = **new_binding**(); *chan* = **duplicate**(); *chan* = **global_commons**()
    - working with *channels* to tuple space regions

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)

Architecture

- topology
- security
- communication & coordination
- software installation & execution
- data storage & management

Software Installation & Execution

- installation & execution rights governed by capabilities
- 3 classes of deployment:
  1. script submitted by general public
     - single Ruby or Perl script
     - runs in extremely restricted language-specific sandbox
     - executed immediately; no permanent installation
     - rate & resource limited
     - no possible access to files
     - similar to Scriptroute; want Scriptroute compatibility layer
     - jobs submitted through central CGI hosted at CAIDA
  2. singleton tool
  3. tool bundle: extension of system

Software Installation & Execution

- 3 classes of deployment:
    1. script submitted by general public
    2. singleton tool
        - single script or executable
        - temporarily installed in a jail and executed once
            - *once* doesn't mean short-lived
        - can access resources with appropriate capabilities
            - including input & output data files
    3. tool bundle: extension of system

Software Installation & Execution

- 3 classes of deployment:
  1. script submitted by general public
  2. singleton tool
  3. tool bundle: extension of system
     - bundle of files: scripts, executables, shared libraries, and static data
     - temporarily/permanently installed
     - executed any number of times on demand
     - optionally registered as a service
     - optional enforced access control and resource limiting
     - optionally in jail

Software Installation & Execution

- terminology: *m-tool* -- a measurement tool, referring generically to script/tool/tool bundle

Software Installation & Execution

- execution vs. measurement
  - *execution*: starting a process
  - *measurement*: performing some task upon request
  - for tools like `traceroute`: execution = measurement
    - user executes command; command performs measurement and exits
  - useful to separate *measurement* from *execution*
    - *execution* requires a high privilege, but *measurement* should not
    - use **measurement servers** to separate measurement from execution
    - implementing measurement servers is easy and natural under Ark
      - server loop:
        1. accept request over tuple space
        2. perform measurement
        3. write result to tuple space or file

Architecture

- topology
- security
- communication & coordination
- software installation & execution
- data storage & management

Data Storage & Management

- goals: **security** and **simplicity**
  - Principle of Least Privilege
  - data integrity & confidentiality
  - prefer simple file-oriented storage mechanisms
    - **eschew databases**: could have, but want to keep deployment footprint small (on underpowered machines) and management complexity low
- approach:
  - use capabilities for fine-grained access control
  - store bulk measurement data in local files and transfer files regularly to central repository
  - use tuple space for modest amounts of data
    - results of immediately-executed one-off measurements
    - summary statistics of long-running measurements

Status

- implemented Ark’s tuple space in Ruby
  - implemented Ruby client binding to tuple space
- no other Ark component implemented yet or planned for short term
- **highest priority**: working on ***conservative upgrade*** of skitter infrastructure
  - replace with tuple space + scamper + misc tools for now
  - working on tools
    - to control scamper from tuple space
    - to have more dynamic destination lists
      - e.g., manage teams of monitors probing every /24
  - Matthew Luckie making improvements to scamper and writing tool to “sort” scamper traces into files for download

Status

- scamper
  - active measurement tool like skitter developed by Matthew Luckie
  - primary topology tool in Ark
  - better than skitter -- supports:
    - IPv4 & IPv6
    - TCP-, UDP-, and ICMP traceroutes
    - ping
    - path MTU discovery
    - fine-grained multiplexing of destination lists
    - programmatic control via socket
    - *warts* format files with more information than *arts++* files
      - cycle start & end markers
      - measurement metadata (e.g., probing parameters)

Status

- hardware expansion of infrastructure
  - starting July 2006, CAIDA assumed operational stewardship of the machines of the National Laboratory for Advanced Network Research (NLANR)
    - NLANR officially ended in June 30, 2006
    - currently decommissioning 170 boxes of NLANR’s Active Measurement Project (AMP)
    - will transition several dozen AMP boxes to Ark infrastructure, increasing our international coverage by 20 countries that never had skitter monitor
      - will also gain IPv6 connectivity

Thanks!

ark-info@caida.org