Resilient Information Architecture Platform for Smart Grid

Gabor Karsai (Vanderbilt)

Project Overview

RIAPS Project Summary

- Goal: To create an *open source software platform* to run Smart Grid applications and demonstrate it through selected applications. A software platform defines:
  - Programming model (for distributed real-time software)
  - Services (for application management, fault tolerance, security, time synchronization, coordination, etc.)
  - Development toolkit (for building and deploying apps)
- Uniqueness:
  - Focus on distributed applications - not only on networking
  - Focus on resilience – services for fault recovery
  - Focus on security – maintain confidentiality, integrity, availability

Project Summary - Motivation

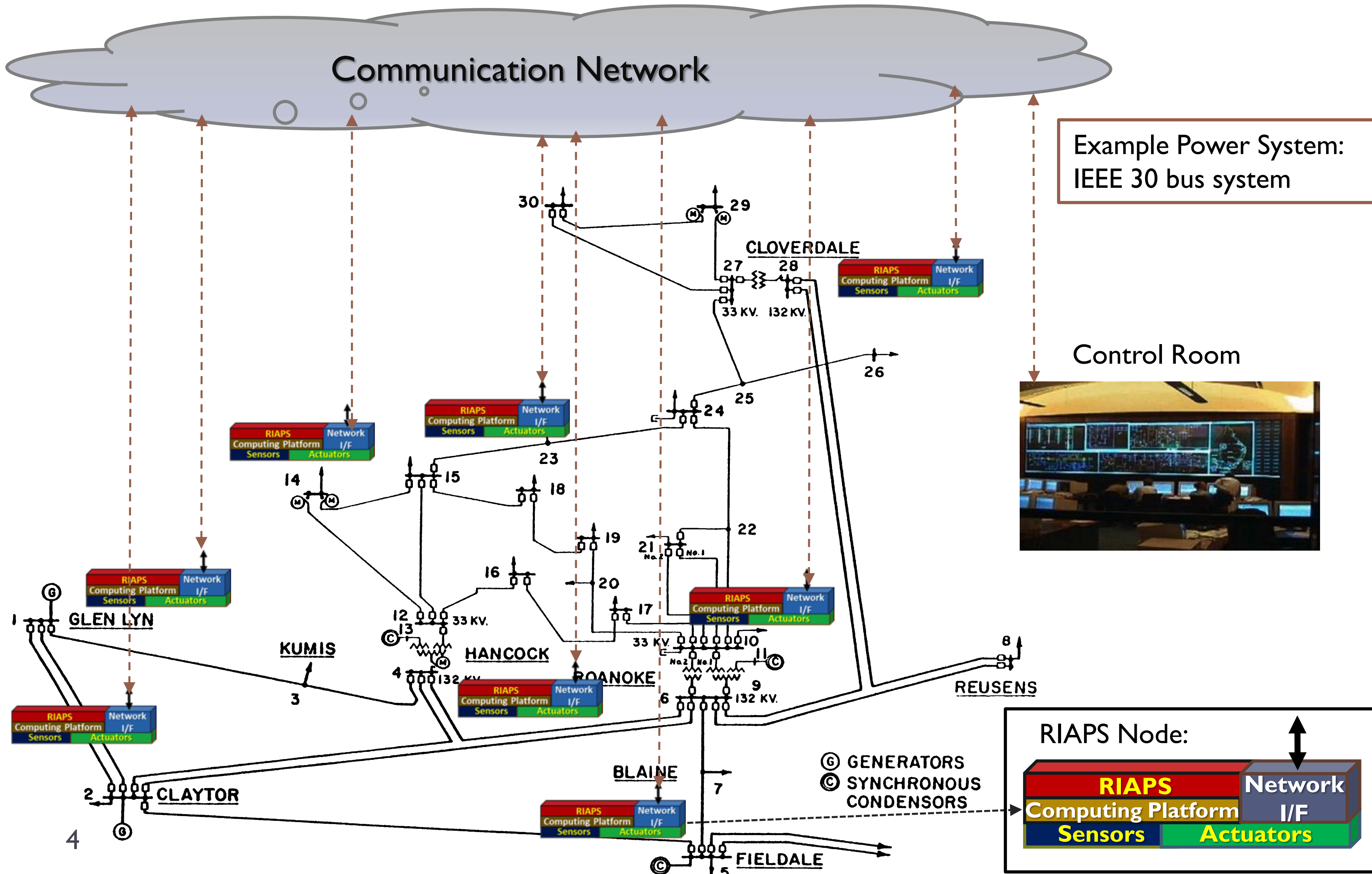
Example Power System:
IEEE 30 bus system

Control Room

GLEN LYN
KUMIS
HANCOCK
ROANOKE
CLOVERDALE
REUSENS
BLAINE
CLAYTOR
FIELDALE
33 KV.
132 KV.

Ⓖ GENERATORS
Ⓒ SYNCHRONOUS CONDENSORS

Project Summary - Motivation



RIAPS: Run-time Platform

Applications

Remedial Action Scheme | Microgrid Management | State Estimation | Data Analytics | Energy Management | Distributed SCADA | ....

Component Framework

- Component Interactions — Component Messaging
- Lifecycle Management — Initialize, Start, Stop, Checkpoint, Destroy
- Component Scheduling — Event/Time-triggered
- Language Run-time — C/C++, etc.
- Resource Management — Resource Management Service
- Security — Access Control, Secure Communications, Secure Information Flows
- Fault Management — Fault Management Service
- Logging — Logging Service
- Persistence — Persistence Service

Platform Managers

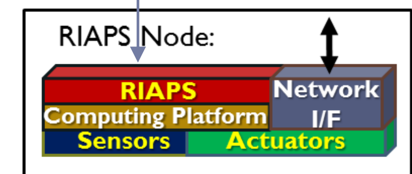
- Application Manager — Application Management and Deployment Service
- Coordination Manager — Distributed Coordination Service
- Discovery Manager — Broker Service
- Time Manager — Time Synchronization Service
- Resource Manager — Resource Management Service
- Device Manager — Device Interface Service
- Fault Manager — Fault Management Service
- Security Manager — Security Management Service
- Log Manager — Logging Service
- Persistence Manager — Persistence Service

OS Kernel

Hardware Platform

Device Interfaces (Sensors/Actuators/Communications/GPS/...) | Network Interface(s) | Storage

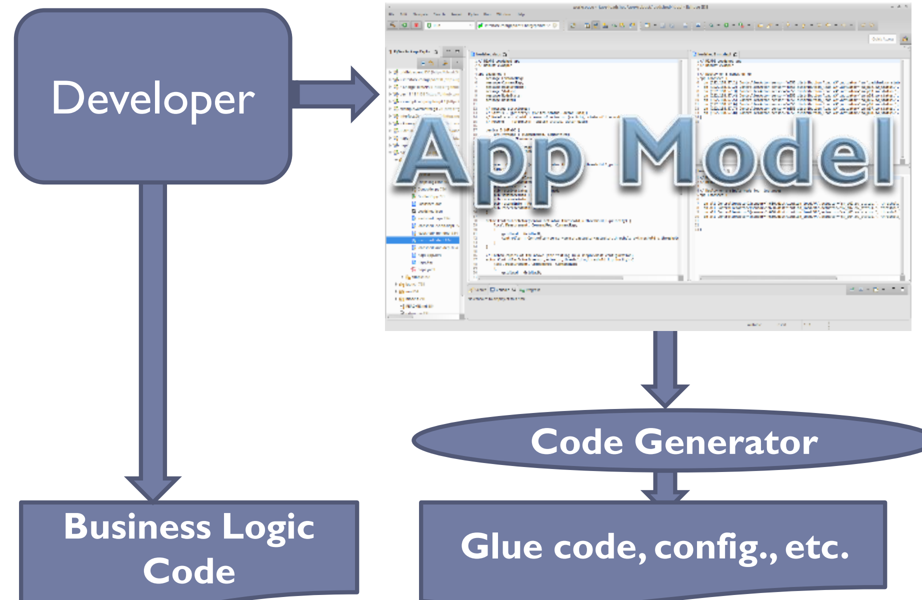
RIAPS Node:

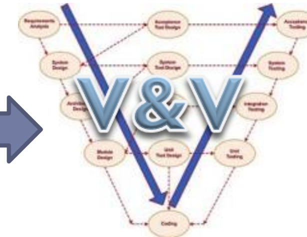


RIAPS Elements: Run-time Platform

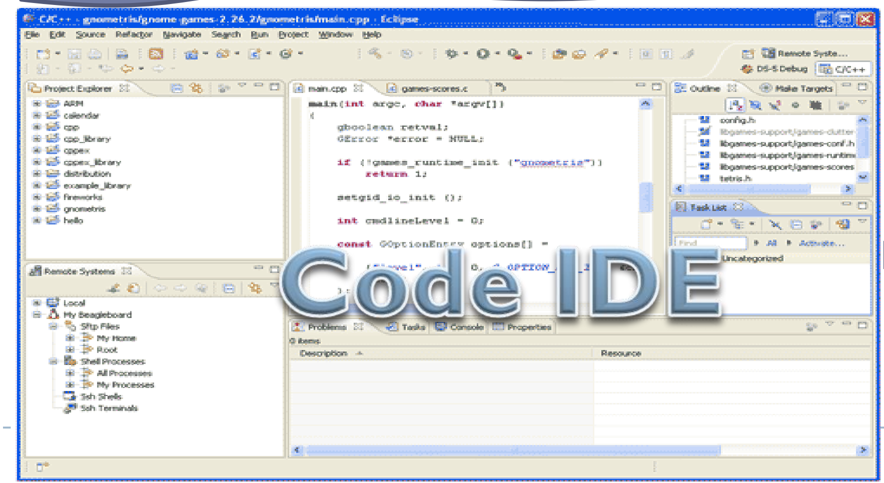
- Software Component Framework:
  - Supports componentized, distributed, real-time apps
  - Core elements: messaging, scheduling, resource / fault management, logging, lifecycle, security, persistence
- Software Platform Managers:
  - Provide global services available to all apps
  - Core elements: discovery, app management, resource / fault management, logging, distributed coordination, time synchronization, security, device interface management, persistence

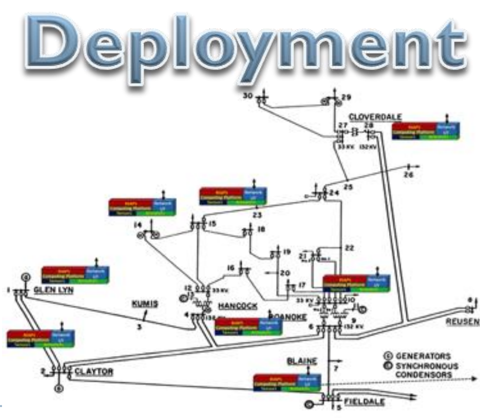
RIAPS Elements: Development tools





The MDE augments the IDE-based development process with a DSML that is used in verification and in generating infrastructure code for the applications.

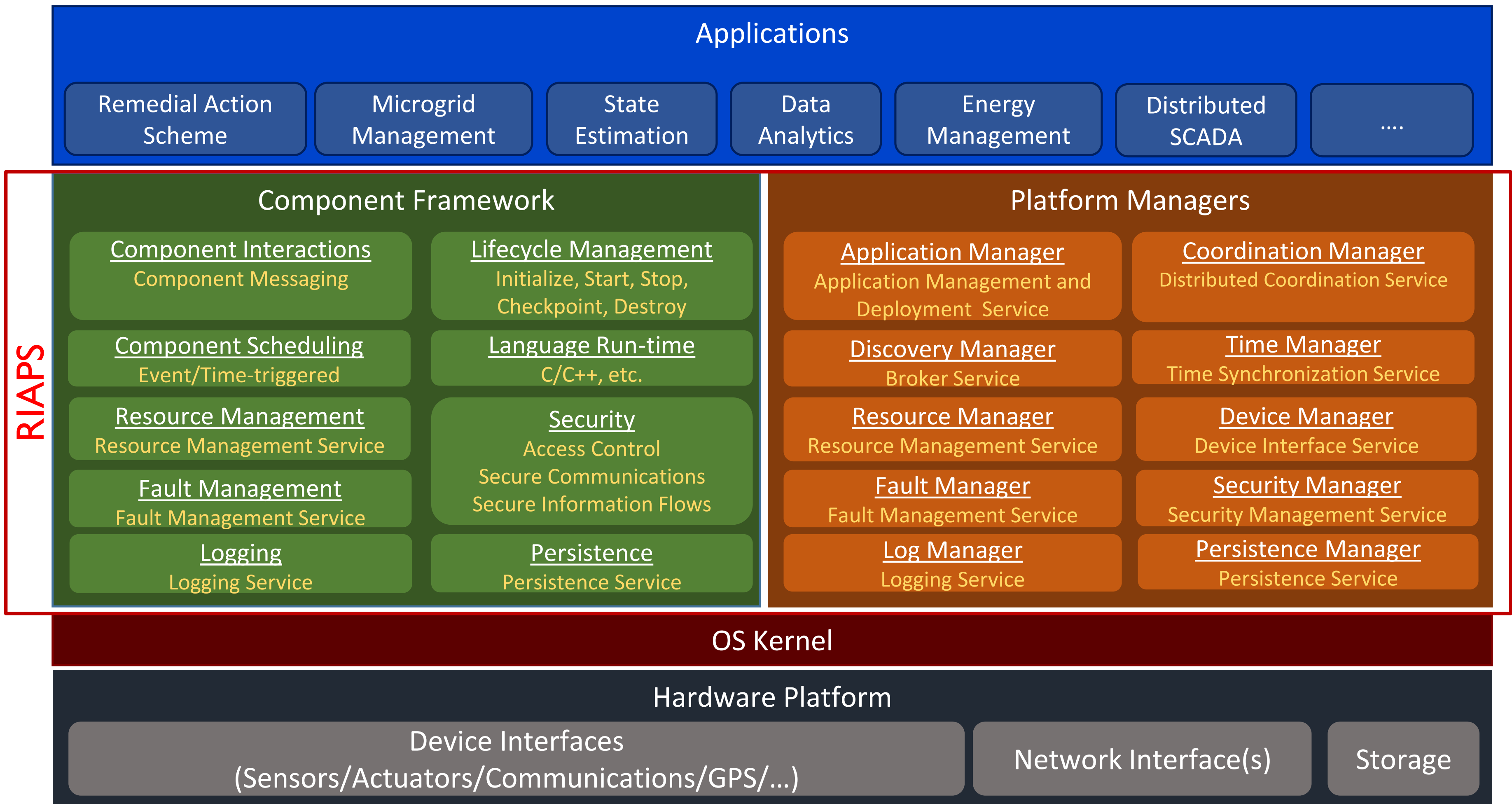




Platform Details

VU

Design: Overall Architecture

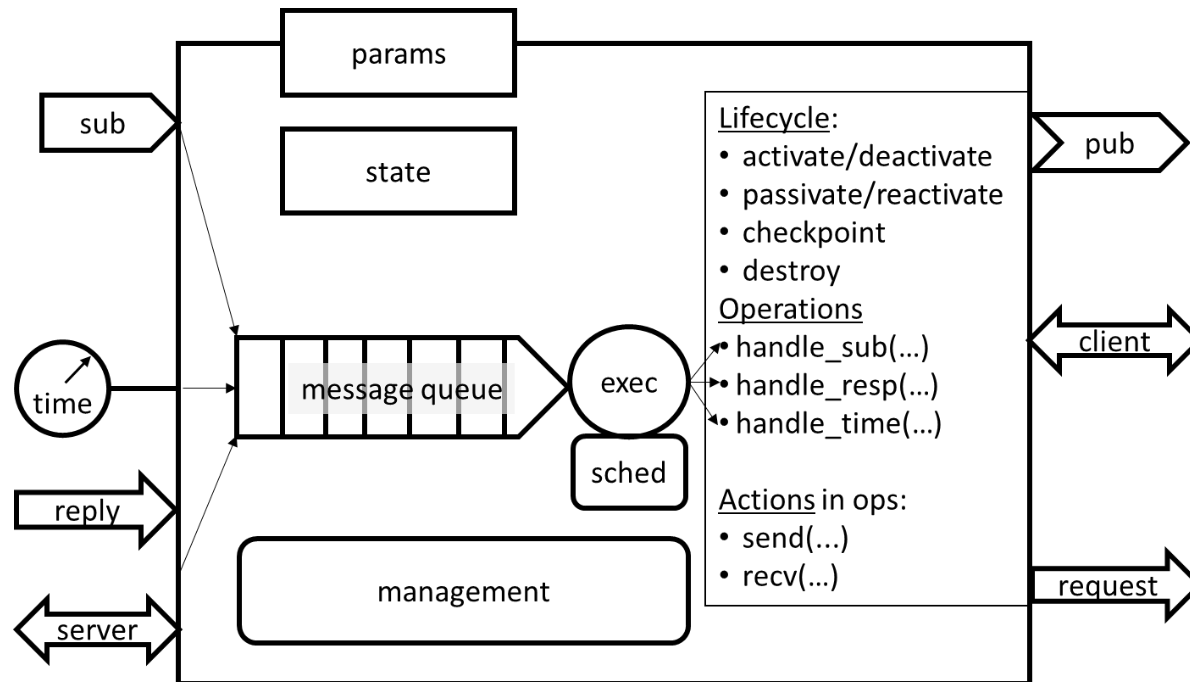


Software Platform

- Basic capability:
  - Software component framework for distributed apps
  - Core platform services: deployment, discovery, devices
  - Domain-specific language for system modeling ; generators
  - Example distributed app: microgrid controls, remedial action scheme
- Enhanced capability:
  - Distributed coordination
  - Time-sensitive messaging
  - Resource management
  - Fault management
  - Example distributed app: transactive energy

Software Platform Details
Component-based development

- Anatomy of a component



subscribe port
timer port
reply port
server port

publish port
client port
request port

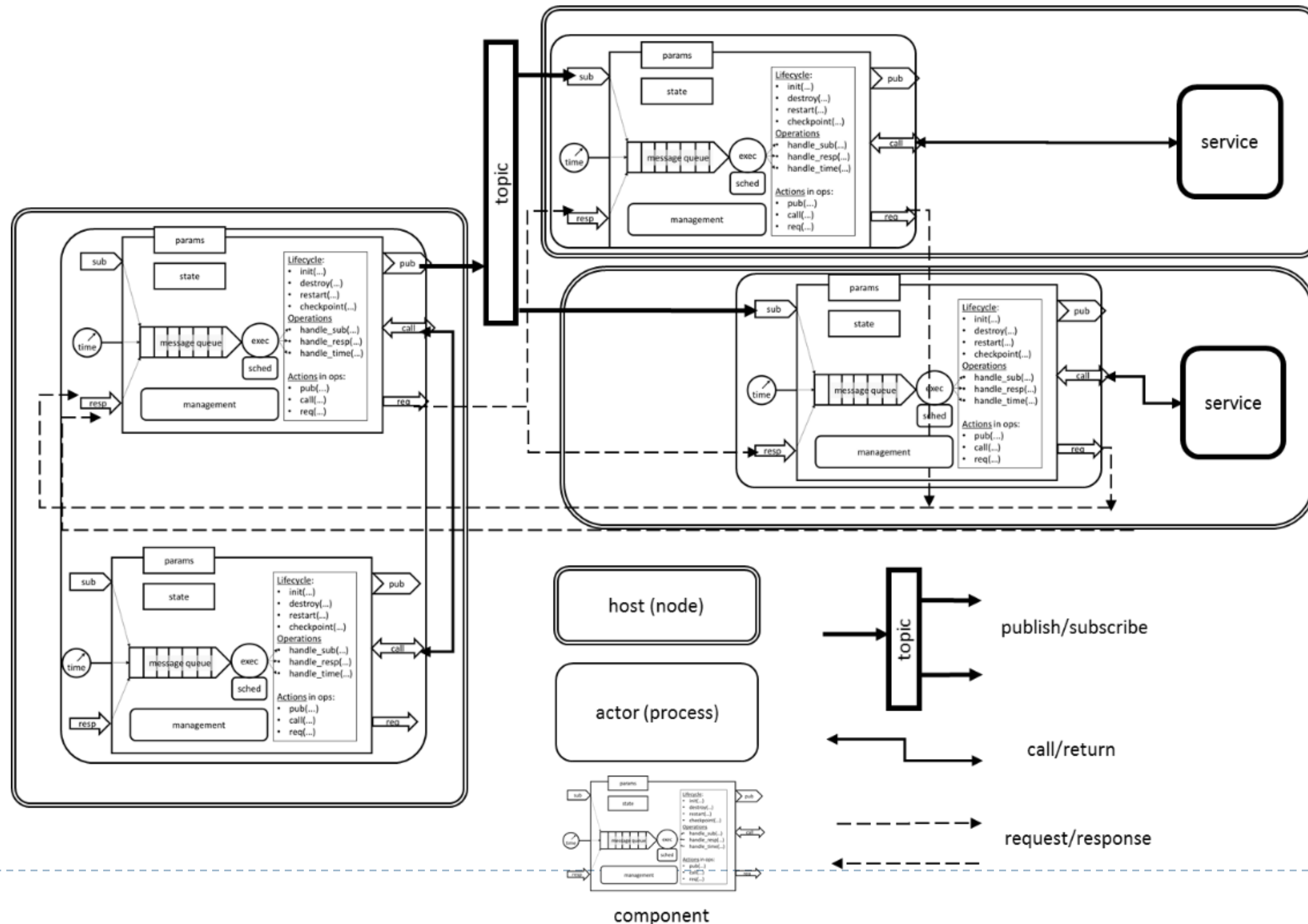
Component can be:
- Event-triggered
- Time-triggered

Components interact via messages

Components have a single execution thread

Software Platform Details
Component-based development

- Anatomy of an application



Software Platform Details
Component-based development

- RIAPS Component Model - Rules
  - Each component runs in its own, *singleton* 'executor' thread
  - Component operations executed *one-by-one* by that thread
  - Components interact with each other using messages *only*
- Rationale
  - Multi-threaded code is complicated, tricky to write, and possibly dangerous
  - App developing power system engineers should not have to learn multithreading

The component model is lightweight – minimal overhead is imposed by the framework. Component/component communication is done by background threads.

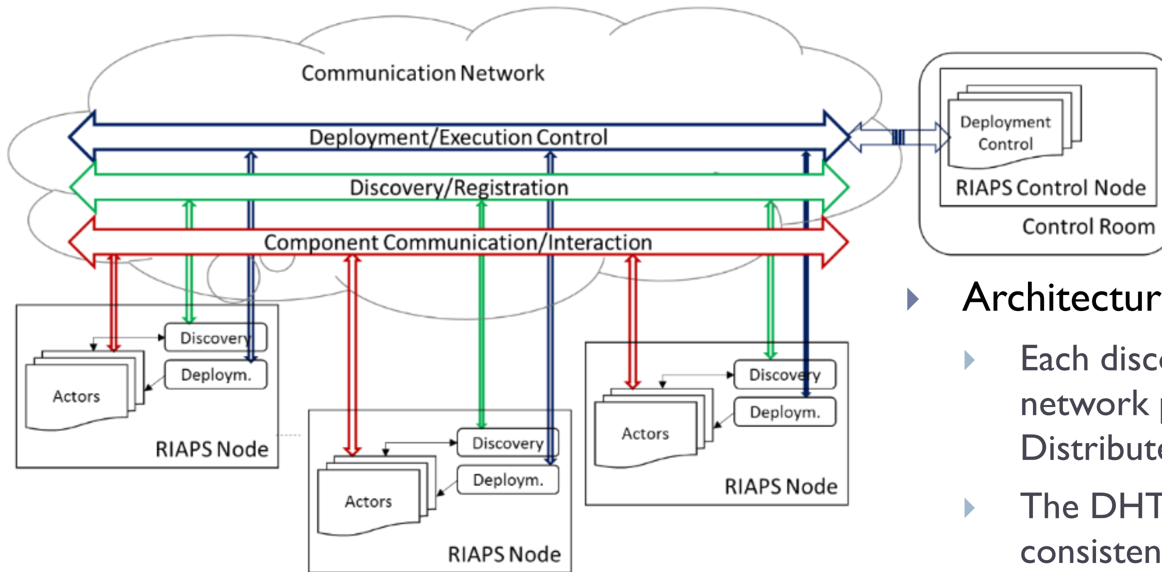
Software Platform Details Component-based development

- Consequences
  - Component is single-threaded
    - No data is shared among components – everything is shared via messages
    - Component internal (state) data does not have to be protected with locks
    - Component developer does not have to (should not) write multi-threaded code
  - Component scheduling discipline (including prioritization of operations) can be controlled on a per-component basis
  - Components are executed concurrently (on the same node) or in parallel (on different nodes of the network)
    - There is concurrency and parallelism *among* components but not *within* components
- Where this approach does NOT work: I/O, asynchronous ops
  - Asynchronous ops and I/O needs to 'connect' to the RIAPS component execution model
  - Solution: 'Devices' (instead of 'components') that *can* be multi-threaded – but for expert developers only!

A 'device component' can encapsulate a complex, multithreaded application/system

Software Platform Details
Discovery service

- Discovery service: To help components find each other



- When a component *provides* a service it registers that with the discovery service
- When a component *requires* a service it looks it up via the discovery service

- Architecture:
  - Each discovery service instance on the RIAPS network participates in a peer-to-peer, Distributed Hash Table (*opendht*)
  - The DHT takes care of maintaining eventual consistency across the nodes of the network
- Advantages:
  - No centralized broker, changes eventually propagate to all participants
  - Robust to node and network failures
- Disadvantages:
  - Content (i.e. registrations) need to be periodically refreshed

Software Platform Details
Development languages

Language	+	-
Python	Easy to learn and use Powerful libraries Interactive development tools	Performance penalty Dynamic typing makes code susceptible to latent flaws
C++	Performance	Hard to learn to use it well Code is more complex, hard to write
Cython	Performance Access to powerful libraries	More complex than Python Requires learning – not widely used

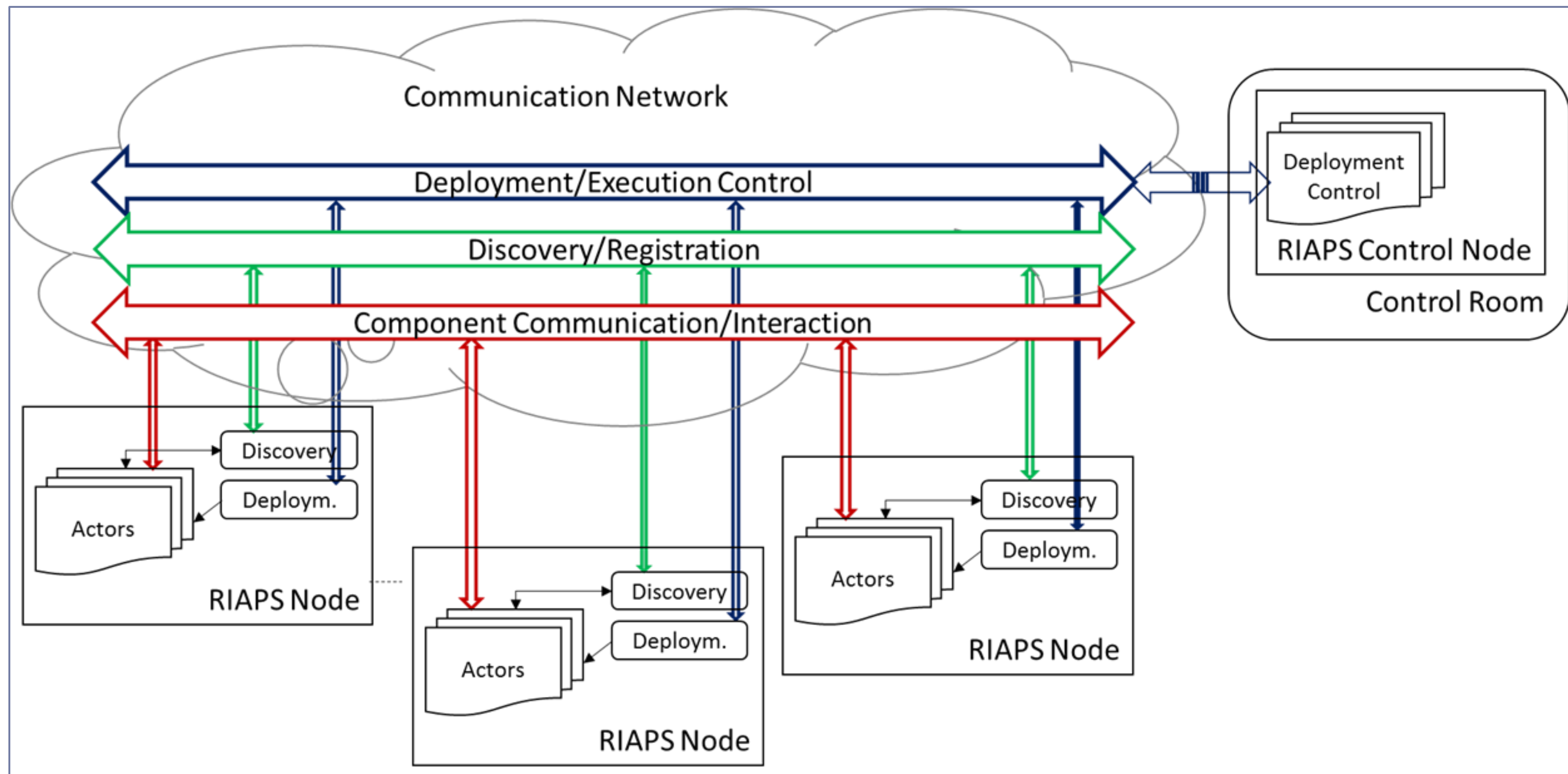
Details:
- Core framework in Python
- Compiled C++ components are loaded into the Python framework
- Other languages can be used for implementing components (e.g. rust)

Recommendations:
1. Prototype algorithms and apps in Python, use fast libraries if needed (e.g. numpy)
2. If better performance is needed, port app to Cython (if learning curve is not an issue)
3. If very high performance is needed, port app to C++

Other implementation choice: Simulink/Stateflow
- Uses the Matlab-generated C code

Software Platform Details
Deployment process

- Assumptions:
  - RIAPS nodes can join and leave the network at *any time*
  - RIAPS nodes are *managed* from a central place (control room)



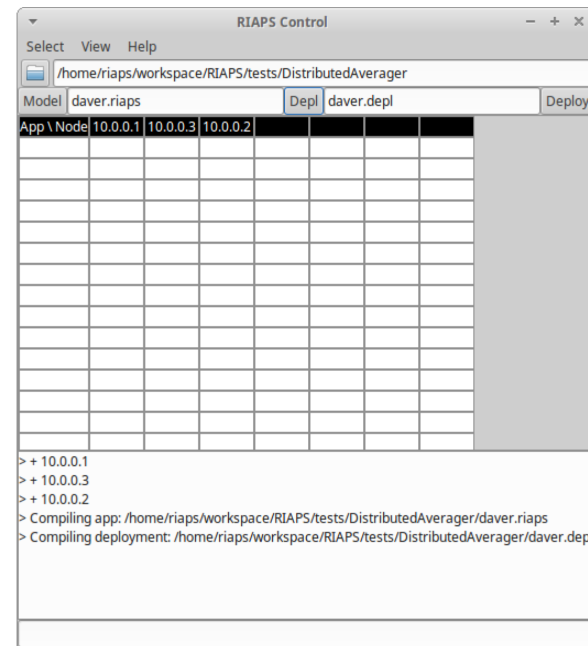
Software Platform Details
Deployment process

- Solution
  - RIAPS Control (the deployment control program running on the control node) maintains a dynamic list of RIAPS nodes.
    - It can also query and track the state of individual nodes.
  - When a RIAPS node joins the network, it registers itself with RIAPS Control, when the RIAPS node disconnects from the network, it is automatically dep-registered from the RIAPS Control
  - RIAPS Control downloads apps to the RIAPS nodes using secure FTP and instructs the node's deployment manager to start the application
- Challenges:
  - Fault management – detecting and recovering from RIAPS node faults at any time during this process
  - Fine-grain management of the deployed application - group control, visibility into the status of the nodes and applications

Software Platform Details
Deployment process

- Solution
  - RIAPS Control (the deployment control program running on the control node) maintains a dynamic list of RIAPS nodes.
    - It can also query and track the state of individual nodes.
  - When a RIAPS node joins the network, it registers itself with RIAPS Control, when the RIAPS node disconnects from the network, it is automatically dep-registered from the RIAPS Control
  - RIAPS Control downloads apps to the RIAPS nodes using secure FTP and instructs the node's deployment manager to start the application
- Challenges:
  - Fault management – detecting and recovering from RIAPS node faults at any time during this process
  - Fine-grain management of the deployed application - group control, visibility into the status of the nodes and applications

App model selected, 3 target nodes connected

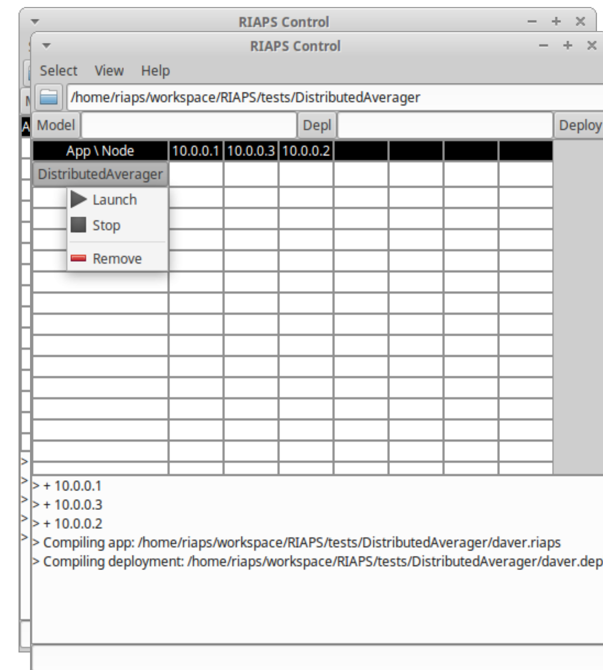


Software Platform Details
Deployment process

- Solution
  - RIAPS Control (the deployment control program running on the control node) maintains a dynamic list of RIAPS nodes.
    - It can also query and track the state of individual nodes.
  - When a RIAPS node joins the network, it registers itself with RIAPS Control, when the RIAPS node disconnects from the network, it is automatically dep-registered from the RIAPS Control
  - RIAPS Control downloads apps to the RIAPS nodes using secure FTP and instructs the node's deployment manager to start the application
- Challenges:
  - Fault management – detecting and recovering from RIAPS node faults at any time during this process
  - Fine-grain management of the deployed application - group control, visibility into the status of the nodes and applications

App model selected, 3 target nodes connected

App deployed, ready to launch



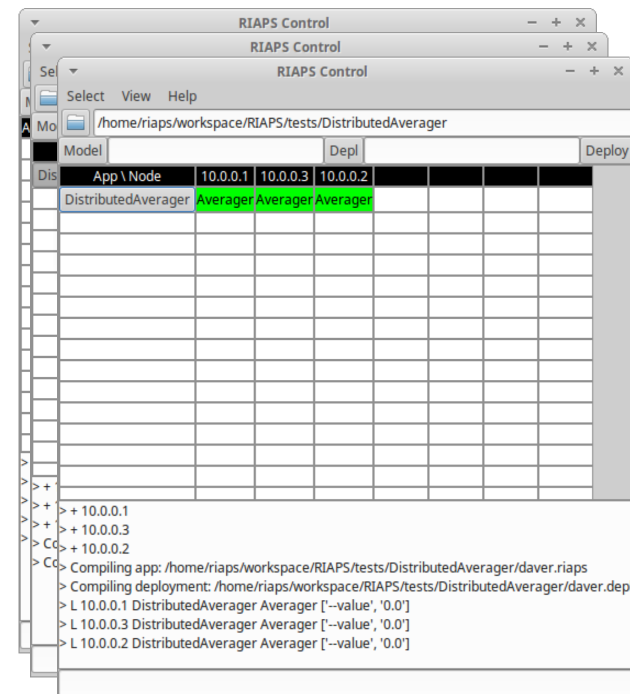
Software Platform Details
Deployment process

- Solution
  - RIAPS Control (the deployment control program running on the control node) maintains a dynamic list of RIAPS nodes.
    - It can also query and track the state of individual nodes.
  - When a RIAPS node joins the network, it registers itself with RIAPS Control, when the RIAPS node disconnects from the network, it is automatically dep-registered from the RIAPS Control
  - RIAPS Control downloads apps to the RIAPS nodes using secure FTP and instructs the node's deployment manager to start the application
- Challenges:
  - Fault management – detecting and recovering from RIAPS node faults at any time during this process
  - Fine-grain management of the deployed application - group control, visibility into the status of the nodes and applications

App model selected, 3 target nodes connected

App deployed, ready to launch

App running on 3 nodes



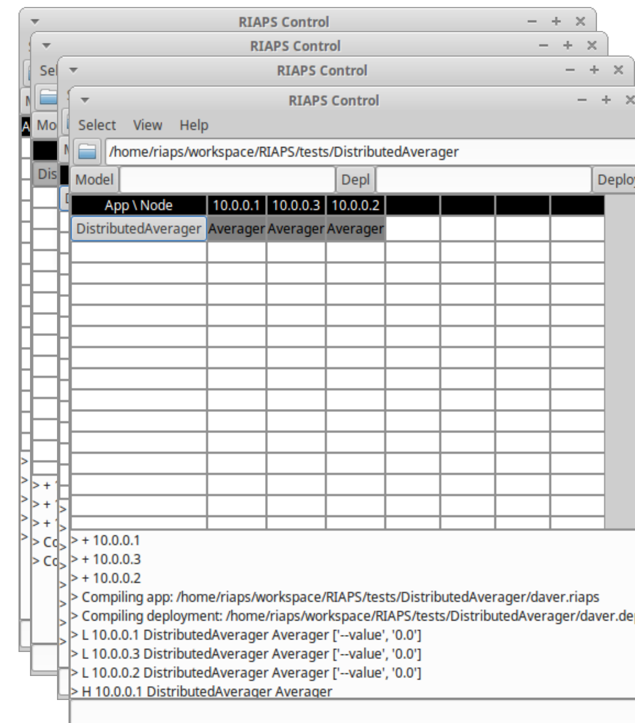
Software Platform Details Deployment process

- Solution
  - RIAPS Control (the deployment control program running on the control node) maintains a dynamic list of RIAPS nodes.
    - It can also query and track the state of individual nodes.
  - When a RIAPS node joins the network, it registers itself with RIAPS Control, when the RIAPS node disconnects from the network, it is automatically dep-registered from the RIAPS Control
  - RIAPS Control downloads apps to the RIAPS nodes using secure FTP and instructs the node's deployment manager to start the application
- Challenges:
  - Fault management – detecting and recovering from RIAPS node faults at any time during this process
  - Fine-grain management of the deployed application - group control, visibility into the status of the nodes and applications

App model selected, 3 target nodes connected

App deployed, ready to launch

App running on 3 nodes



Software Platform Details
Deployment process

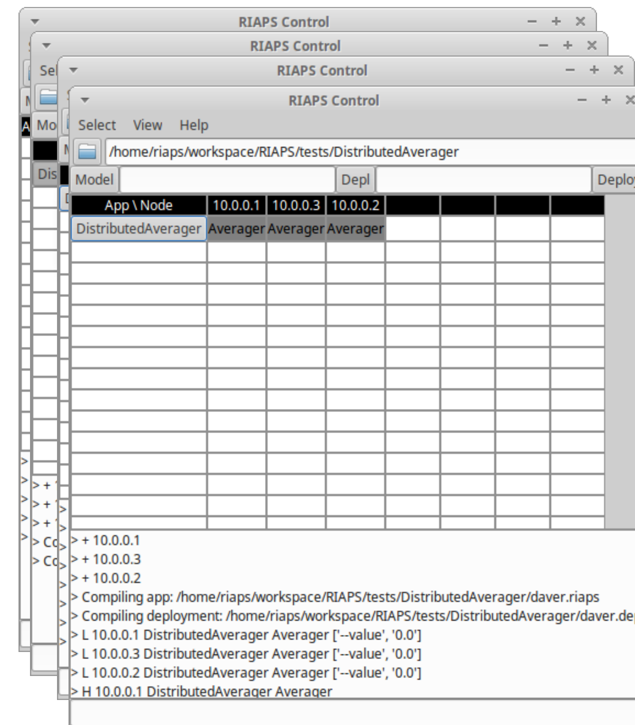
- Solution
  - RIAPS Control (the deployment control program running on the control node) maintains a dynamic list of RIAPS nodes.
    - It can also query and track the state of individual nodes.
  - When a RIAPS node joins the network, it registers itself with RIAPS Control, when the RIAPS node disconnects from the network, it is automatically dep-registered from the RIAPS Control
  - RIAPS Control downloads apps to the RIAPS nodes using secure FTP and instructs the node's deployment manager to start the application
- Challenges:
  - Fault management – detecting and recovering from RIAPS node faults at any time during this process
  - Fine-grain management of the deployed application - group control, visibility into the status of the nodes and applications

App model selected, 3 target nodes connected

App deployed, ready to launch

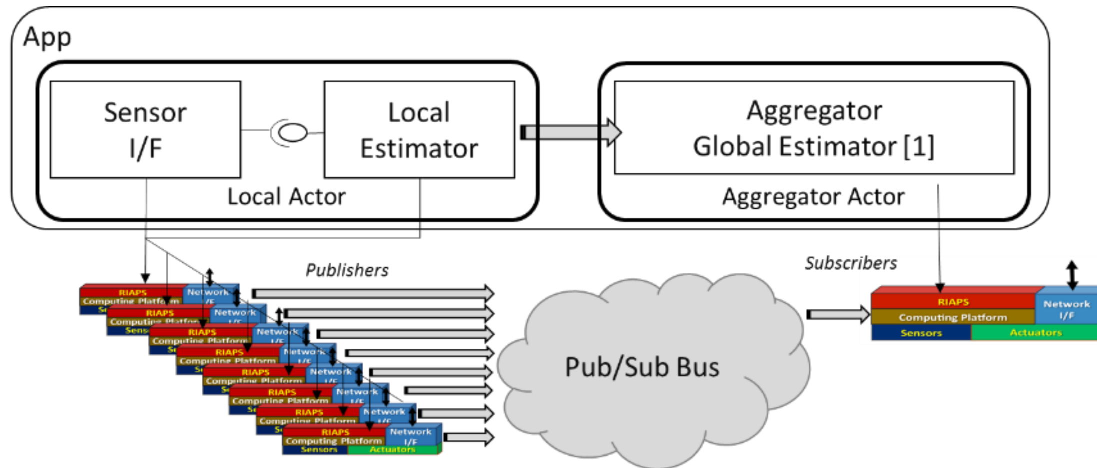
App running on 3 nodes

App halted



Software Platform Details Component-based development

- Example app:



- Components: Sensor, Local Estimator, Global Estimator
  - Each runs in its own thread
  - Interact via messages only
- Actors: Local Actor and Aggregator Actor
  - Local Actors are deployed on multiple nodes
  - Aggregator actor is deployed on a single node

- Concurrency:
  - Sensor and Local Estimator run in separate threads but concurrently (possibly on different cores) - pipelining
  - Local Aggregator Actors and run on different nodes, in parallel
  - Aggregator Actor (and its Global Estimator component) runs on a separate node, in parallel with everything else

Software Platform Details
Models and code in applications - Example

```
// RIAPS Sample

app DistributedEstimator {
    // Message types used in the app
    message SensorReady;
    message SensorQuery;
    message SensorValue;
    message Estimate;

    // Sensor component
    component Sensor {
      timer clock 1000;                               // Periodic timer trigger to trigger sensor every 1 sec
      pub ready : SensorReady ;                       // Publish port for SensorReady messages
      rep request : ( SensorQuery , SensorValue ) ;   // Reply port to query the sensor and retrieve its value
    }

    // Local estimator component
    component LocalEstimator (iArg,fArg,sArg,bArg) {
      sub ready : SensorReady ;                       // Subscriber port to trigger component with SensorReady messages
      req query : (SensorQuery , SensorValue ) ;      // Request port to query the sensor and retrieve its value
      pub estimate : Estimate ;                       // Publish port to publish estimated value messages
    }

    // Global estimator
    component GlobalEstimator (iArg=123,fArg=4.56,sArg="string",bArg=true) {
      sub estimate : Estimate ;                       // Subscriber port to receive the local estimates
      timer wakeup 3000;                              // Periodic timer to wake up estimator every 3 sec
    }

    // Estimator actor
    actor Estimator {
       local SensorReady, SensorQuery, SensorValue ;    // Local message types
       {  // Sensor component
          sensor : Sensor;
          // Local estimator, publishes global message 'Estimate'
          filter : LocalEstimator(iArg=789,fArg=0.12,sArg="text",bArg=false);
       }
       filter.query = sensor.request;                   // Sensor and local estimator are connected
    }
    actor Aggregator (posArg,optArg="optString") {
       {  // Global estimator, subscribes to 'Estimate' messages
          aggr : GlobalEstimator(iArg=posArg,sArg=optArg,bArg=true);
       }
    }
}
```

Software Platform Details
Models and code in applications

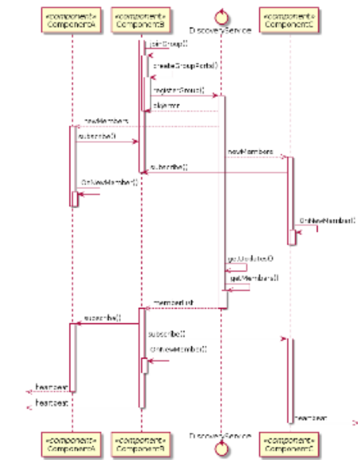
- Application Model specifies:
  - Message types (with data fields)
  - Component types (with ports)
    - Each 'input' port requires an associated message handler – i.e. the component operation
    - Publish/subscribe ports have single message types
    - Request/reply and client/server ports have pairs of message types (for request -> reply types)
  - Actors (with components)
    - List components
  - Message flows between components and actors are inferred from the message types
    - Connections are automatically established ate deployment time
  - Components and actors can have parameters
    - Formal and actual, incl. default values
- Generated from the model:
  - Python/C++ code skeleton for component(s) + JSON file to configure run-time system
- Application code (by developer):
  - C++: Extend skeleton code with 'business logic' for the code
  - Python: Implement the component class and its operation
- Deployment Model: Specifies how to deploy actors on the RIAPS nodes
  - On all / selected nodes, with concrete parameter values

Software Platform

- Basic capability:
  - Software component framework for distributed apps
  - Core platform services: deployment, discovery, devices
  - Domain-specific language for system modeling ; generators
  - Example distributed app: microgrid control
- Enhanced capability:
  - Distributed coordination
  - Time-sensitive messaging
  - Resource management
  - Fault management
  - Example distributed app: transactive energy

Distributed coordination services

- Motivation:
  - Need for precisely defined and verified services to support apps consisting of activities interacting via a network
- Group membership:
  - An app component can dynamically create/join/leave a *group* of entities of the same app. It can also send/receive messages within the group, and be informed about changes in the group membership
- Leader election:
  - A group can have a *leader*: an 'elected' component that makes global decisions. Leaders are elected through an automated process, and communication to/from the leader is possible.
- Consensus:
  - Group members can get participate in a *consensus* process that reaches agreement over a value via a special protocol.
- Time-coordinated control action:
  - Group members can use a combination of the above three features to agree on a control action that is executed at a scheduled point in time in the future



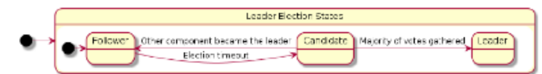
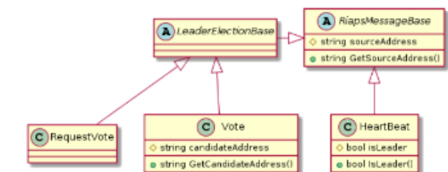
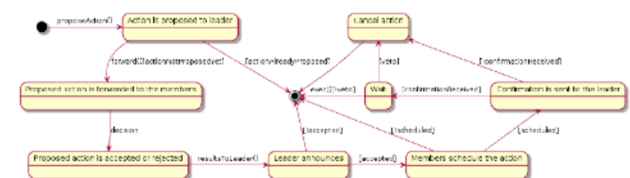


Figure 6: States of the leader election





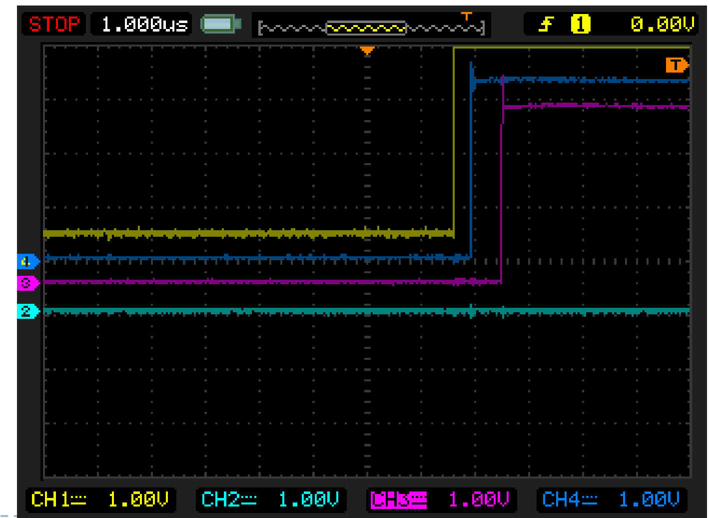
Distributed coordination services

- Test example – Synchronized action
  - 3 nodes form a group – *Group formation*
  - Group elects a 'leader' node – *Leader election*
  - Every 5 second nodes propose a future time value for control action that the group votes on – *Consensus*
  - Agreed-upon time value is used to schedule a control action in the future that is executed on all nodes when the time arrives – *Time-coordinated control action*

Accuracy of time-coordination is better than 5 microseconds (given the time-synchronization)

Time synchronization:

- Master clock GPS (fallback NTP)
- Clock distribution: IEEE 1588
- Node clock deviation < 10 usec (on LAN)



Time-Sensitive Messaging

Modeling language extension:

```
// Sensor component
component Sensor {
   …
   pub ready : SensorReady timed;            // Publisher of SensorReady messages
   rep request : ( SensorQuery , SensorValue ) timed; // To query the sensor
}
// Filter component
component Filter () {
  sub ready : SensorReady timed ; // Subscriber of SensorReady messages
  req query : (SensorQuery , SensorValue ) timed ; // To issue queries for the
sensor
  …
}
```

Motivation: app needs to know how long it took to transfer a message in the network.

Sender-side:

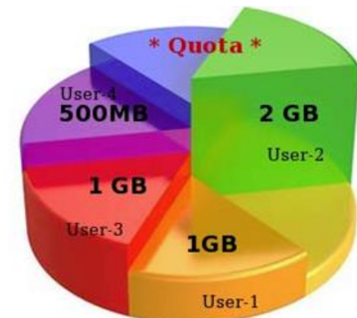
- Timestamping is automatic

Receiver-side:

- get_sendtime(): when message was sent
- get_recvtime(): when message was received

Resource management - Approach

- Resource: memory, CPU cycles, file space, network bandwidth, (access to) I/O devices
- Goal: to protect the 'system' from the over-utilization of resources by faulty (or malevolent) applications
- Use case:
  - Runaway, less important application monopolizes the CPU and prevents critical applications from doing their work
- Solution: model-based quota system, enforced by framework
  - Quota for application file space, CPU, network, and memory + access rights to I/O devices + response to quota violation – captured in the application model.
  - Run-time framework sets and enforces the quotas (relying on Linux capabilities)
  - When quota violation is detected, application actor can (1) ignore it, (2) restart, (3) react to by freeing resources.
    - Detection happens on the level of actors
    - App developer can provide a 'quota violation handler'
    - If actor ignores violation, it will be eventually terminated



Resource Management Implementation

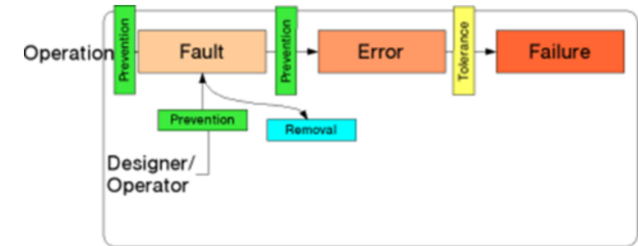
		Detection	Enforcement	Mitigation
CPU Utilization	Soft limit	cgroups	cgroups automatically adjusts priority of actor as needed	Automatic
	Hard limit	Process monitor	Monitor signals process	App-provided handler
Memory footprint	Soft limit	Process monitor	Monitor signals process	App-provided handler
Disk space	Hard limit	Quota system for files	Quota system for files; monitor signals process	App-provided handler
Network utilization	Hard limit	Network 'tc'	'tc' caps the network bandwidth a process can use; monitor signals process	App-provided handler
Operation deadlines	Soft limit	Track time spent in op	Monitor signals process	App-provided handler
Unexpected termination of operation	N/A	Exception handler	Exception handler signals component	App-provided handler

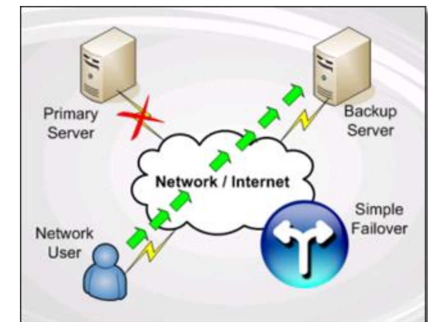
Notes:

- 'cgroups' (for 'control groups') is an advanced Linux feature that is widely used to control resource usage of programs (CPU, memory, network, etc.)
- 'tc' (for 'traffic cop') is an advanced Linux feature to control the network packet handling in the OS. One can limit the rate (number of bytes sent over a time interval) and 'tc' will delay or drop packets of the limit is exceeded.

Fault Management

- Assumption
  - Faults can happen anywhere: application, software framework, hardware, network
- Goal
  - RIAPS developers shall be able to develop apps that can recover from faults anywhere in the system.
- Use case
  - An application component hosted on a remote host stops permanently, the rest of the application detects this and 'fails over' to another, healthy component instead.
- Philosophy:
  - The platform provides the mechanics, but app-specific behavior must be supplied by the app.





Fault management

Principles:

- Application actor termination will be detected by the deployment manager, and the actor will be restarted per the application model
- Application resource violation and operation deadline violation will be logged, and application notified
- RIAPS services will be automatically restarted if they crash
- The RIAPS node's core operating system will be configured in an automatic restart mode
- Network connection health will monitored via heartbeat messages and application notified
- Failed application deployment will be detected at the control node
- Loss of connectivity to the control node will detected, logged, and re-establishment of connection attempted

System-level Fault Management Implementation

Fault location	Error	Detection	Recovery	Mitigation
App flaw	actor termination	deplo detects via netlink socket	(warm) restart actor	call term handler; notify peers
	unhandled exception	framework catches all exceptions	if repeated, (warm) restart	call component fault handler; notify peers about restart
	resource violation	framework detects	if restarted →	call app resource handler notify peers
	- CPU utilization	soft: cgroups cpu		tune scheduler
		hard: process monitor	if repeated, restart	notify actor/ call handler
	- Memory utilization	soft: cgroups memory (low)		notify actor/ call handler
		hard: cgroups memory (critical)	terminate, restart	call termination handler
	- Space utilization	soft: notification via netlink		notify actor/ call handler
		hard: notification via netlink	terminate, restart	call termination handler
	- Network utilization	via packet stats	if repeated, (warm) restart	notify actor/ call handler notify peers about restart
	- Deadline violation	timed method calls	if repeated, restart	notify component / call handler
	app freeze	check for thread stopped	terminate, restart actor	notify component; call cleanup handler; notify peers restart
	app runaway	check for method non-terminating	terminate, restart actor	notify component; call cleanup handler; notify peers about restart

System-level Fault Management Implementation

Fault location	Error	Detection	Recovery	Mitigation
RIAPS flaw	internal actor exception	framework catches all exception	terminate with error; warm restart	call term handler;
	disco stop / exception	deplo detects	deplo (warm) restarts disco	if services OK, upon restart restore local service registrations
	deplo stop	systemd detects	restart deplo	(cold) restart disco ; restart local apps
	deplo loses ctrl contact	deplo detects	NIC down -> wait for NIC up; keep trying	
System (OS)	service stop	systemd detects	systemd restarts	clean (cold) state
	kernel panic	kernel watchdog	reboot/restart	deplo restarts last active actors
External I/O	I/O freeze	device actor detects	reset/start HW; device - specific	inform client component
	I/O fault	device actor detects	reset/start HW; device - specific	log, inform client component
HW	CPU HW fault	OS crash	reset/reboot	systemd → deplo
	Mem fault	OS crash	reboot	systemd → deplo
	SSD fault	filesystem error	reboot/fsck	systemd → deplo
Network	NIC disconnect	NIC down		notify actors/call handler
	RIAPS disconnect	framework detects RIAPS p2p loss	keep trying to reconnect	notify actors/call handler ; recv ops should err with timeout, to be handled by app
	DDoS	deplo monitors p2p network performance		notify actors/call handler

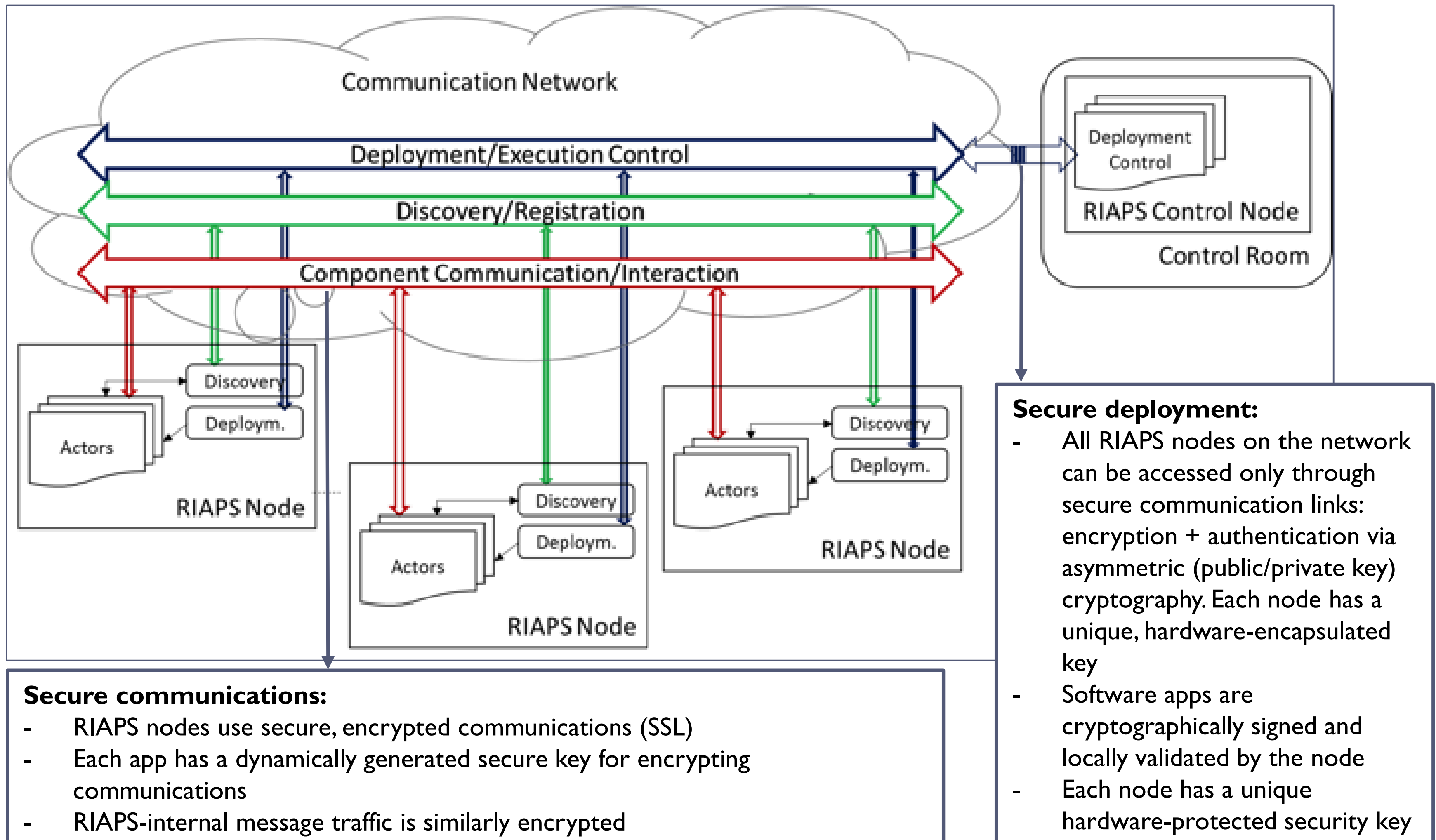
Some implementation details for fault management

- Application restart:
  - When application actors are started, they are registered in a fault-tolerant local database. Upon crash of the actor, RIAPS services, and the RIAPS node itself, the deployment automatically manager restarts all application actors that were running previously (i.e. they are in the database).
- Detection of loss of network connectivity:
  - The Network Interface Card (NIC) is continuously monitored -- if the 'carrier' is lost and/or restored, the running applications are informed.
- Handling remote application crashes:
  - When a RIAPS node starts, its deployment manager attempts to join a peer-to-peer network of RIAPS nodes.
  - When an application is deployed and an application actor starts, its peers within the RIAPS network will be informed.
  - If an application actor has crashed and/or restarted, its peer application actors are informed about the loss and reappearance of the actor.
  - The above mechanism allows implementing apps that are aware of status changes of their actors

Software Platform
Basic Security: Threat model

Type of threat	Possible harm	Protection is needed for
Malicious network devices	Observe, possibly modify or disrupt network traffic	Availability of resources; confidentiality and integrity of communications
Malicious applications	Interfere with operations, exhaust resources, or physically damage the node or the connected power system	Confidentiality and integrity of communications, availability of and control over physical resources
Distributed Denial of Service (DDoS) attack	Core applications of the platform are unable to operate	Platform services and remotely deployed and controlled applications
Malicious application actors	Unauthorized access to configuration and operational data of another application	Confidentiality of data

Software Platform
Basic Security: Design



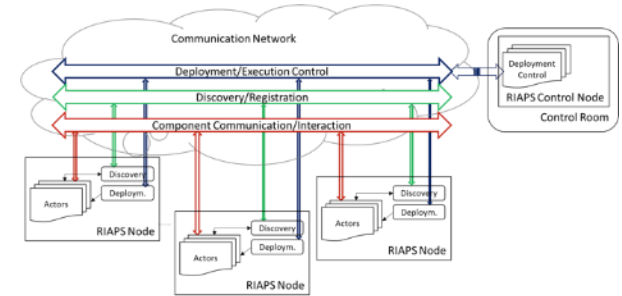
Software Platform
Basic Security: Design

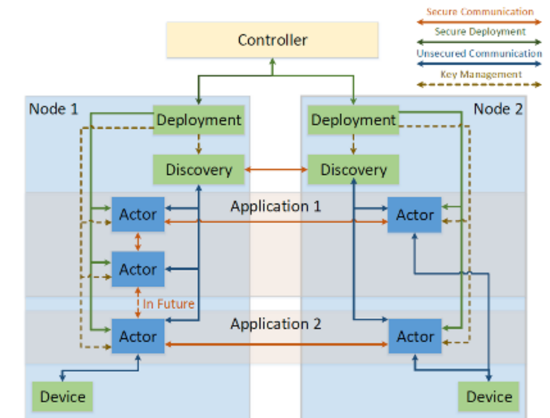
How are security concerns addressed?

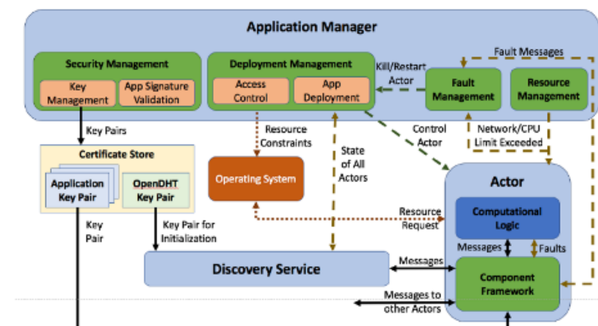
	Secure deployment	Secure communications
Confidentiality	SSL: Encrypted, secure communications between control room and nodes Cryptographically signed apps	SSL: Encrypted, secure communications between control room and RIAPS nodes Each app has a generated crypto key for encrypting all communications
Integrity		
Authenticity		
Availability	Detection of network impairments Multiple network links (fallback)	Informing apps about impairment Autonomous operation of nodes and apps is expected

Management:

- Node-local security manager to keep and manage security keys
- Deployment manager cryptographically validates apps during deployment, assigns keys to app actors
- Discovery service uses system-specific keys for inter-node communications)







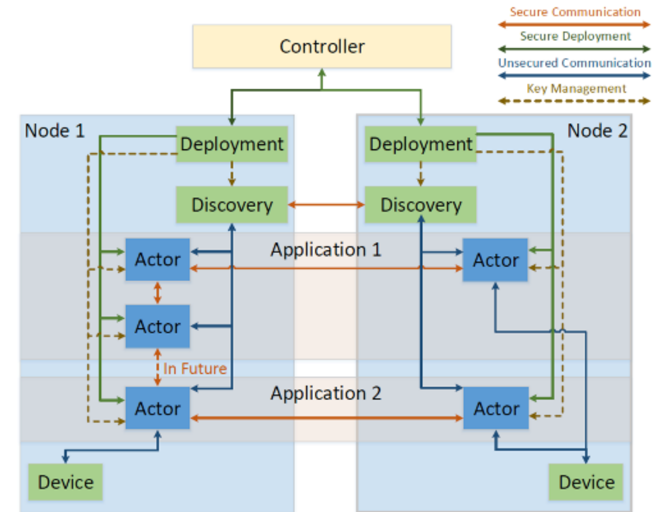
Software Platform
Advanced Security: Design

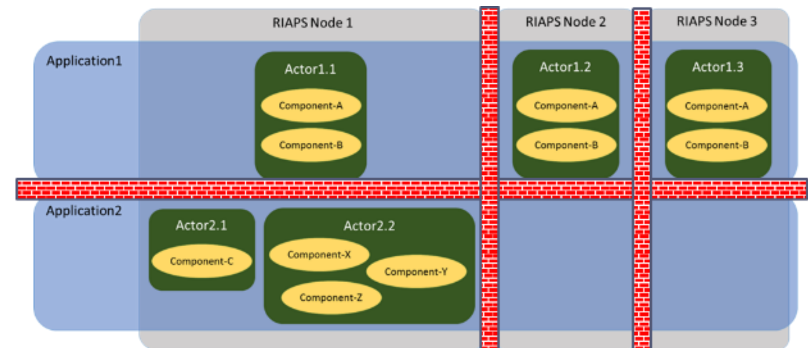
- Advanced app-level security
  - Isolation: Apps are to be isolated from each other using standard Linux access control facilities: a unique, new user id is generated at deployment time and the app runs under this id – with restricted privileges.
  - Access control: Uses *AppArmor* – a lightweight Mandatory Access Control (MAC) service in Linux that provides resource protection including: file access (read, write, link, lock), library loading, execution of applications, coarse-grained network (protocol, type, domain), Linux capabilities, coarse owner checks (task must have the same euid/fsuid as the object being checked), file system mounting, named sockets, abstract and anonymous sockets, DBus API (path, interface, method), signals, ptrace services. Configuration of these protections will be done using the application model.
    - Security configuration will be enforced by the deployment manager when the application is deployed and launched.

Software Platform
Advanced Security: Implementation

Secure information flow enforcement:

- **Messages:** All actor-actor communication is encrypted using ZeroMQ's Elliptic Curve (EC) encryption mechanism that uses dynamically generated security keys.
- **Files (and other resources):** Applications are 'firewalled' from each other using Linux's AppArmor Security Module. The riaps_actor instances are not permitted to access any files except the ones in their home and in the /tmp folder.





Software Platform
Advanced Security: Implementation

- *Support for Advanced Capability: supporting security features for application.*

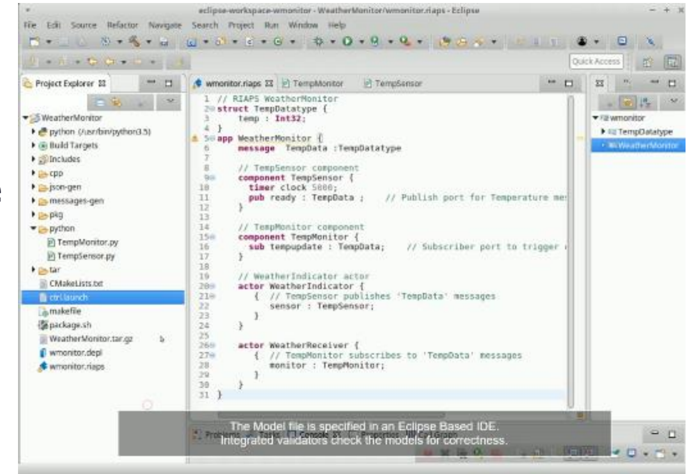
Deployment model constrains what network resources an app actor can access.

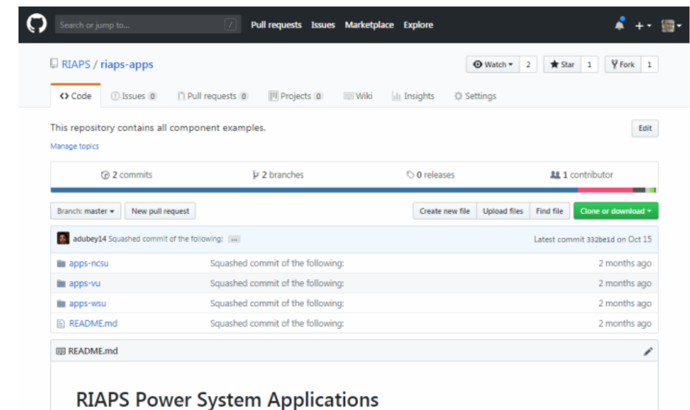
These rules are strictly enforced by the Linux firewall system.

```
app DistributedEstimator {
 host 192.168.57.1 {
  network any; // Actors on this host may connect to any Internet node
 }
 host all {
  network dns; // all hosts may connect to the domain name service
  }
 host 192.168.57.3 {
   network 192.168.1.1; // … may connect to 192.168.1.1
 }
 on all Estimator;    // Estimator actor deployed on all nodes
 on (192.168.57.1) Aggregator(posArg=123); // Aggregator on
                                           // 192.168.57.1 only
}
```

Support for engineering, commissioning, and maintenance of applications

- Envisioned app development process
  - Developers use a state-of-the-art IDE for development
  - The development process is model-driven – code skeletons and configuration artifacts are automatically generated from models
    - See RIAPS DSML for applications and deployment
  - Developers use a source code management system:
    - All files are version controlled, all versions are preserved
    - SCM allows 'tagging' the file versions such that a unique configuration can always be retrieved from the database
- Current RIAPS IDE and tooling
  - Eclipse IDE with RIAPS plugin (DSML generator)
  - git version control system
    - Examples hosted on https://github.com/RIAPS





Support for engineering, commissioning, and maintenance of applications

- Solution
  - When application is deployed, the SCM is tagged with a dynamically generated 'tag' - a special code that identifies a specific state of the source code base, i.e. the versions that were used in the deployment (*)
  - The application files are packed into a compressed package that includes metadata: the code base tag, and other information about the deployment (*)
  - The package is cryptographically signed and encrypted(*)
  - The package is transferred to the target nodes
  - The received package is validated, decompressed, metadata is stored (*)
  - The application is started, operated etc.

Why?
Application code actually deployed can be traced back to the source code in the SCM

Metadata shows the provenance of the code

Encryption protects from tampering with the code

If validation fails, package was modified

RIAPS Data Models

RIAPS and Data (1)

- RIAPS is not a 'program' (a software application), rather a complete software platform to build applications
- RIAPS has many internal data models, not exposed to the applications
  - Architecture Modeling Language
    - Describes the components and the 'wiring' of a distributed app
    - https://github.com/RIAPS/riaps-pycom/blob/develop/src/riaps/lang/riaps.tx
    - Textual syntax, parsed and converted into JSON
  - Deployment Modeling Language
    - Describes the deployment aspects: hosts (with firewall permissions) and actor/process deployment on a network
    - https://github.com/RIAPS/riaps-pycom/blob/develop/src/riaps/lang/depl.tx
    - Textual syntax, parsed and converted into JSON
  - Deployment manager/Actor messages
    - For all interactions between the 'deplo' and app 'actors'
    - https://github.com/RIAPS/riaps-pycom/blob/develop/src/riaps/proto/deplo.capnp
    - Defined in capnproto IDL, compiled into marshaling code for C++/used in Python
  - Discovery service/Actor messages
    - For all interactions between the 'disco' and app 'actors'
    - https://github.com/RIAPS/riaps-pycom/blob/develop/src/riaps/proto/disco.capnp
    - Defined in capnproto IDL, compiled into marshaling code for C++/used in Python

RIAPS and Data (2)

- RIAPS internal data models, not exposed to the applications
  - App-internal messages
    - ZMQ wrapper over raw bytes of app payload (with opt. timestamp)
    - ZMQ wrapper dynamic group/coordination messages (RAFT, etc.)
  - App deployment package
    - .tgz file, cryptographically signed/encrypted
    - Content:
      - Component code (.so or .py), support libraries, data files
      - App architecture model and deployment model (in .JSON)
      - Firewall configuration, source traceability information
      - EC security keys to encrypt in-app network comms
  - Deployment manager package
    - Public/private keys for protecting 'ctrl'/'deplo' comms
    - EC security keys to encrypt in-deplo network comms
    - Cert for authenticating logins to 'ctrl'

RIAPS and Data (3)

- RIAPS apps define their own data model
  - App message data models:
    - Python: any object (c-pickled)
      - Sender and receiver must agree on data model
    - C++ and Python: Defined in capnproto IDL
      - IDL is translated into C++ classes and marshaling/unmarshaling code
      - IDL can be directly loaded into Python (the component) and used via a dynamically instantiated API
- RIAPS apps can use a configurable logger
  - Based on spdlog, usable from Python or C++
  - Log format is customizable by the app developer
- RIAPS apps can connect to external services
  - Typical approach: RIAPS device component to manage interactions
  - Examples: C37.117, Modbus, ChargePoint (OCPP),
  - Same applies to any other network-accessible resource (e.g. Influxdb)

Summary

Summary

- RIAPS is a *platform* for building distributed apps for Smart Grids
- It has been demonstrated with
  - Microgrid control app
    - Islanding/reconnection, distributed control
  - Remedial action scheme app
    - Generation curtailment and under-frequency load-shedding
  - Transactive energy app
    - Prosumer 'traders' buy and sell energy, use a blockchain to record trades
  - Priority-based load shedding
    - Loads controlled by their own RIAPS nodes that receive 'grid load' information and disconnect/reconnect their nodes according to a pre-defined priority scheme

https://riaps.isis.vanderbilt.edu/

https://riaps.github.io/

https://github.com/RIAPS

https://www.youtube.com/channel/UCwfT8KeF-8M7GKhHS0muawg

RIAPS was made possible by support from the US DOE ARPA-E