A system architecture for future autonomic communication networks

Last lecture

- We introduced the key abstractions for autonomic networking:
  - Information channels
  - Functional Blocks
  - Information Dispatch Points

This is where we want to go to:

- We need:
  - Flexibility
  - Virtualization
  - Autonomicity
- How to implement such a system?
Outline

- vlink
  - Integration in the existing network infrastructure
- System architecture
  - Minmex
- Monitoring Example

The `ana_vlink` subsystem

"Virtualizing connectivity"

Christophe Jelger, UBasel

Motivation

- How to test a new network architecture which is not backward compatible?
  - The code will run on legacy systems (Linux)
  - ANA will co-exist with legacy networking (IP) code
  - We want to inter-connect distant sites
  - We want a flexible way to "wire" ANA networks
  - We want to virtualize resources

... leading to requirements that

- ANA code must not disturb or conflict with legacy networking code (IP)
  - i.e., an Ethernet or IP compartment must not screw up your local IP connectivity and applications.
- The core ANA code must provide a flexible abstraction layer (or subsystem) to emulate low-level connectivity between ANA nodes
So what's the problem?

<table>
<thead>
<tr>
<th>IP</th>
<th>IPv6</th>
<th>ANA</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0x0800)</td>
<td>(0x86DD)</td>
<td>(0x????)</td>
</tr>
</tbody>
</table>

We differentiate ANA traffic from legacy stuff.

Ethernet frame has a type field indicating "what it carries".

Ethernet is not enough ...

<table>
<thead>
<tr>
<th>IP</th>
<th>IPv6</th>
<th>ANA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Compartment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0x????)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UDP (0x????)</td>
</tr>
</tbody>
</table>

We get rid of whatever encapsulation (ETH, GRE, UDP)

Possible to inter-connect distant ANA nodes

to other ANA nodes

and from there "re-start from scratch" for the legacy world inside ANA

Here there could be ETH+IP or whatever new ANA protocols

We get rid of the first Ethernet header (a kind of encapsulation)

Differentiating ANA traffic is not enough

- We want to virtualize "connectivity"
  - i.e., physical NICs must be shared.

We introduce the concept of "virtual links" at the ANA level.
Building your ANA net "on-demand"

System Architecture

MinMEX Elements

- Minimal INfrastructure for Maximal EXtensibility
  - Internal packet dispatch framework
    - Used to create packet processing paths in the MINMEX
    - for example linking an application socket to an interface, or
      "binding" a set of functional blocks.
  - Key-Val Repository
    - A simple tuple storage is provided, akin to a MIB with (one level of) key-value associations. System variables, binding of method names to the function, scratch memory for coordination among extensions, belong here. Also, extensions are listed at this place, as well as any services imported and/or instantiated at run-time.
MinMEX Elements continued

- Bootstrap procedure
  - generic mechanisms that MUST be implemented by all implementations of an ANA node.
  - The bootstrap procedure is used by an ANA node to discover its surrounding and to start running real protocols that live in the playground.
  - “Signaling is out-of-band”: i.e., the bootstrap protocols do not rely or use data paths created and maintained by “real” protocols.

MinMEX Elements continued

- MINMEX controller
  - We keep a minimal and autonomic control entity inside MINMEX to continuously monitor the operation of MINMEX.
    The MINMEX controller performs a continuous assessment of the basic operation of an ANA node. It is truly autonomic and basically performs sanity/health checks of the MINMEX operation.
  - It also monitors the operation of the bootstrap procedure and ensures it is always healthy.

Controller

- There is no independent controller component in the MinMex for now
- However we already have some internal checking mechanisms:
  - IDT checks for deprecated entries
  - Key Val Rep checks for deprecated entries
  - Heart beat mechanism between MinMEX and Bricks in order to free resources in case of failure
Information Dispatch Table

- Stores actions associated to IDPs
- IDPs of all Bricks attached to the MinMex are stored in one single table.
- Some IDT entries are public, i.e., accessible by all Bricks
- Private views possible:
  - Certain entries can only be accessed by members of the view
  - For now a view has only one member, extension to groups possible
- The IDT has garbage collection:
  - Deletes Entries unused for long period
  - Deletes Entries allocated to a Brick on its failure detection
- Possibility of permanent entries (during the Brick’s lifetime)
  - Limitations still to specify..

Shadow Dispatch Table

- The Bricks and the MinMex are in many cases (except Kernel mode) different processes
- Therefore, the MinMex can not map IDP labels directly to actions (i.e., functions) but rather redirect messages between bricks
- The Label->function conversion is done at the receiving Brick
- This Brick maintains a Shadow dispatch table mapping labels to functions

Key Value Repository (KVR)

- One of the main components of the MinMex
- Allows a Brick to publish "functionality" (+ IDPs)
- Entries in the KVR have the following fields:
  - An entry name that must be unique in the repository
  - An unlimited set of keywords
  - An IDP label as the value to be used to reach "publisher"
- The KVR has periodic garbage collection:
  - Deletes Entries unused for long period
  - Deletes Entries mapped to no longer existing IDPs
- Possibility to have permanent entries (during the Brick’s lifetime)
  - Limitations/restrictions to be specified later
Key Value Repository

- Possible requests:
  - Lookup by name:
    - The requesting Brick previously knows the name of the "service" it wants to reach
    - Exact name matching
  - Lookup by keywords:
    - Unlimited set of keywords
    - For the moment the entries returned are the ones that match the entire set of keys
    - Boolean requests on keywords

Message types

- ANA Data/Control messages:
  - They are encapsulated with IDP labels
- ANA Node configuration:
  - Outside ANA World (flexible way to "wire" the ANA Node)
  - Limited command requests and replies between the Brick and the MinMex:
    - Attachment/detachment of Brick from the MinMex
    - Creation/Deletion of IDPs
    - Redirect of IDPs destinations
    - Setting of notification channel
- Notification:
  - Unsolicited information sent from the MinMex to the Brick(s)
  - The information could concern both ANA World and MinMex-Brick liaison

"Inter-process communication"

- Communication between the MinMex and the Bricks can be done by a network protocol or IPC
- So far we have implemented:
  - UDP sockets
    - useful for the ana@Home to allow distant Bricks to join a Presence Point MinMex
  - Unix sockets
  - Named pipes
  - Generic netlink (Kernel Space)
  - Direct function call (Kernel)

Gates Paradigm

- We call communication Gate any way to receive messages. A gate could be:
  - A UDP socket
  - UNIX socket
  - Named Pipe ...
- Both MinMex and Bricks have 2 communication gates:
  - A Data Gate
    - To receive data and notification messages
  - A Control Gate
    - To receive control messages
Gates Paradigm

• For control/data exchange the control/data Gates of the Brick and the MinMex do not need to use the same communication mode (UDP, PIPES...)
• The following case is possible:

Monitoring Example

Current status

• Implementation of some monitoring bricks and two basic scenarios:
  – Capturing
  – Adaptive Packet Sampling
  – System Monitoring

Capturing brick

• Provided functionalities for:
  – Packet capturing
  – Packet inspection (for some protocol header)
  – Configurable in terms of:
    • Filter rules: type of packets to be captured (i.e., ethereal filter rules format – e.g., “ip saddr x.y.z.w …”, or “**”)
    • Capturing state: start, stop, pause
Capturing brick
- Multiple consumer bricks can bind the capturing brick:
  - Specifying the IDPs to which captured packets can be received
  - Which type of packets are requested

Adaptive Sampling brick
- It performs sampling algorithms on packets received from a capturing brick
- It can be dynamically configured in terms of:
  - Sampling algorithm to be performed
    - Although so far only a systematic sampling algorithm is available
  - Sampling rate
    - It can be configured by the monitoring compartment (i.e. external configuration)
    - Automatically, based on some "situational" conditions (i.e. self-configuration)
      - Currently, it is able to adapt the sampling rate in order to keep constant the CPU load of the hosting node

System Monitoring brick
- Its aim is to measure system performance parameters and deliver them to consumer bricks
- By now, it is able to measure node’s CPU load
- A "pluggable" approach easily allows to add new measurement functions (very small API!)

- Different modes to deliver measured data:
  - On request
  - Subscriber mode, based on:
    - Timer
    - Condition (e.g. inform me when CPU > x)
Adaptive Scenario

- The following scenario has been experimented:

Some real traffic traces have been used to test it

Another scenario