Node Architectures & Functional Composition

Prof. C. Tschudin, M. Sifalakis, T. Meyer,
G. Bouabene, M. Monti

University of Basel

Cs321 - HS 2011

Lecture Overview

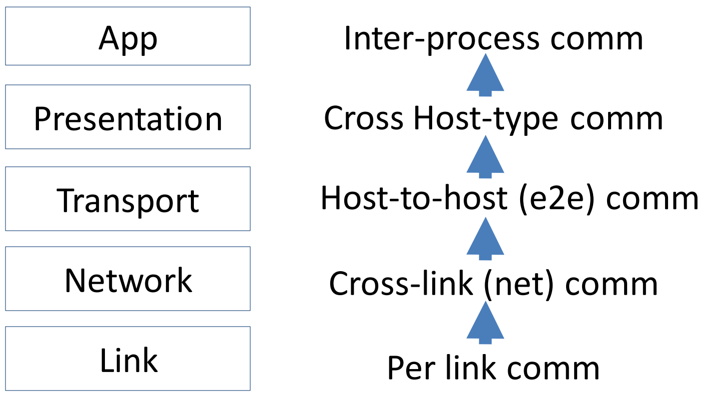
- Architecture engineering for complex systems
  - Component versus layer based engineering
- Execution Models of distributed computation and cooperation
- Engineering of runtime adaptive systems: Some example architectures
  - The Autonomic Network Architecture
  - OMNet++ (!?)
  - Tau

Taking a step back: Software engineering for complex systems

- 6 important principles of complex system design
  - Abstraction
    - Ability to generalise and contextualise functionality
  - Functional Composition/Synthesis
    - Structuring complex systems by combining simpler functions
  - Factorisation of functionality
    - Divide and conquer the problem space
    - Decomposition of complex functions in simpler tasks
  - Modularity
    - Information hiding and separation of function specification from mechanism
  - Virtualisation
    - Federation/Pooling and Distribution of resources
  - Re-use of functionality
    - Use of functionality in different contexts without repetition of mechanism

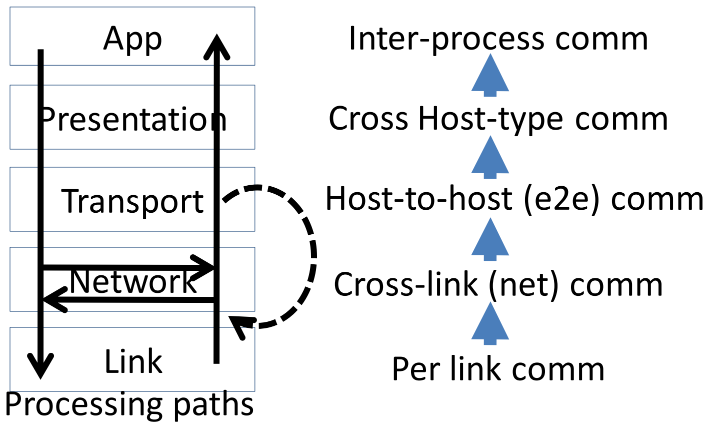
Engineering using Layers

- Single line of Abstraction
  - Sequential/incremental abstraction across layers



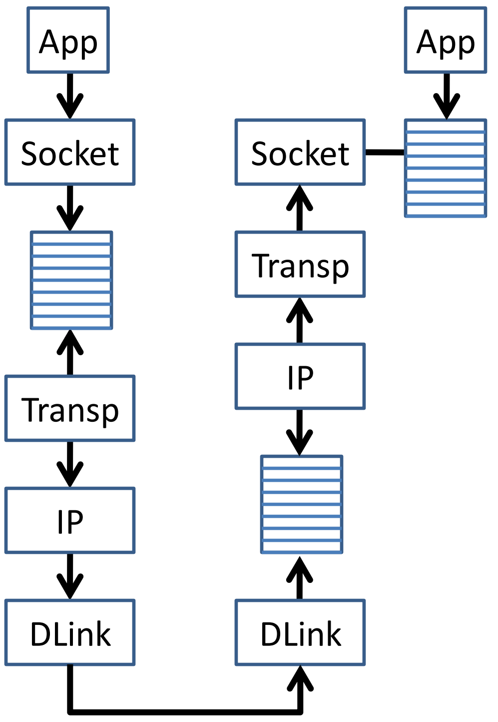
Engineering using Layers

- Single line of Abstraction
  - Sequential/incremental abstraction across layers
- Linear Composition
  - Ordered processing among functions from each layer
  - Internet reality: Potential recursive loops may appear across the processing chain (breaking linearity)
    - E.g. IP in IP, L2PT tunnelling, etc



Engineering using Layers

- Factorisation of functionality
  - Division across layers
  - Nothing implied about intra-layer.
- Modularity at the layer boundaries
  - Internet reality: Extreme opaqueness across layers
    - Exchange of data only, no signalling or control information across layers



Typical implementation of a TCP/IP stack

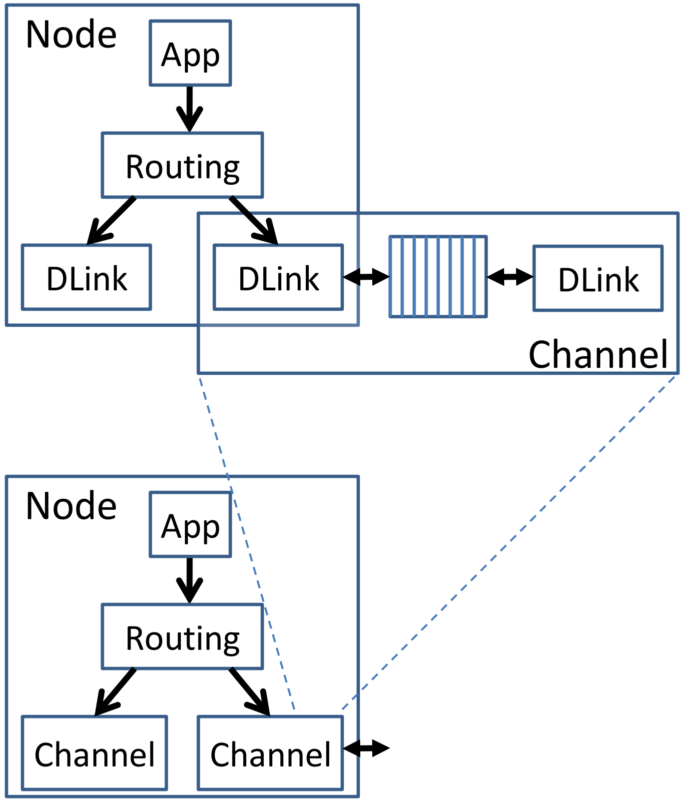
Engineering using Layers

- Virtualisation of layer services at the semantics level
  - E.g. Various Data link technologies pooled together under an Ethernet VLAN interface
  - E.g. Multiple links pooled together as a virtual link (IP tunnel or TCP flow)
- Re-use of functionality
  - No re-use of functionality across layer boundaries. Rather repetitive functionality is often seen!!

Internet node functions often do not follow layer semantics and more often are neither engineered in a layered way

Engineering based on Components

- Abstraction
  - Flexible, for any grouping of functions
  - Polymorphic(!): same group of functions can represent different abstractions
- Composition
  - Linear and non-Linear arrangement of functions (loops)
  - Recursive: Functions containing instances of themselves
- Factorisation of functionality
  - Arbitrary to fit optimisation objectives



Engineering based on Components

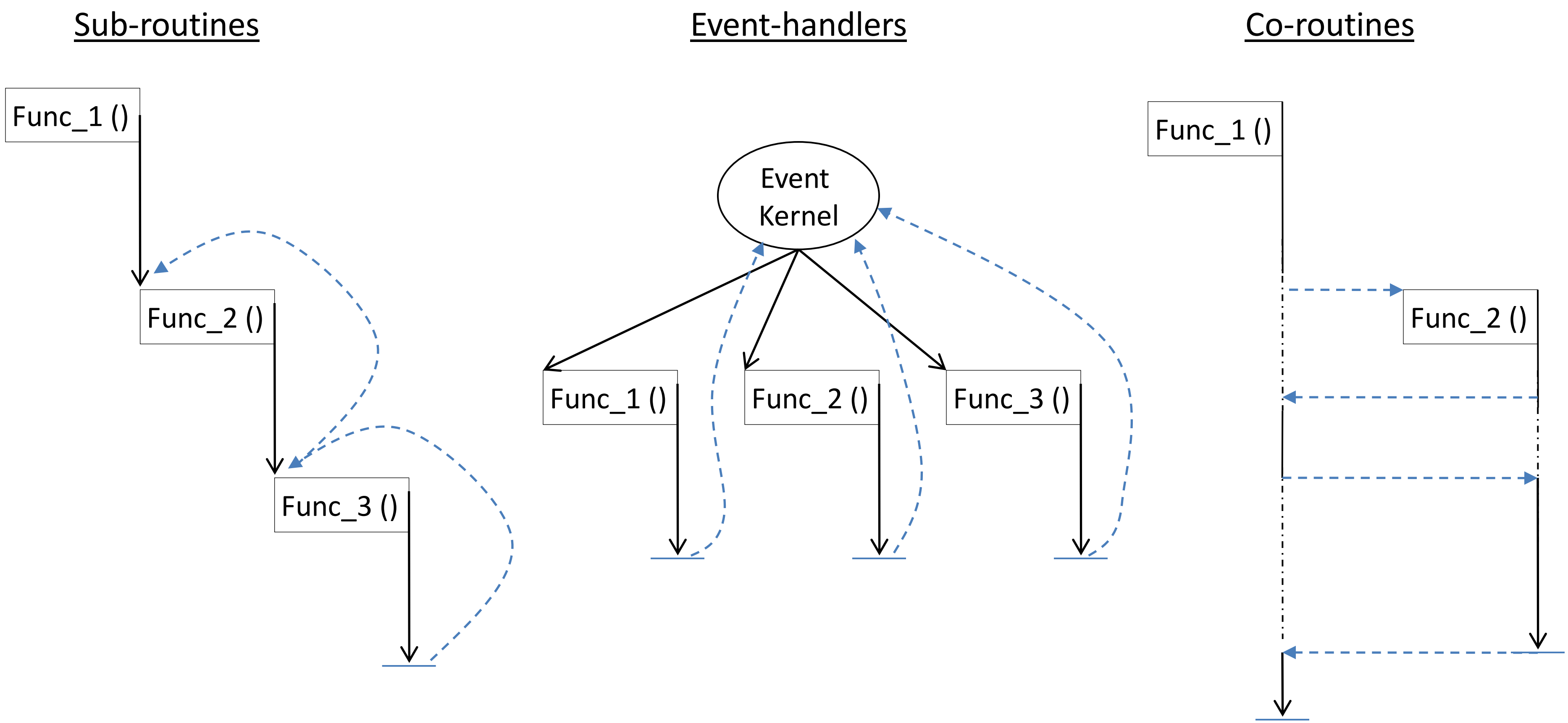
- Modularity
  - Between components as well as between levels of abstraction (compound components)
  - Extensible or multiple interfaces
- Virtualisation at the interface level
  - Homogeneous interfaces allow even heterogeneous resources to be federated
    - E.g. Net storage, disk storage, memory disk and database storage can be pooled and used in a federated way
- Re-use of functionality
  - Same function can be accessed in different contexts
  - Inheritance: Functions derived or extending other functions

Shift in design patterns ?

- Summary comparison of layer-oriented design versus component- oriented design

	Layers	**Components**
Abstraction	Sequential	Polymorphic
Composition	Linear	Linear / Non Linear
Factorisation of functionality	Across layers	Flexible to fit Optimisation obj.
Modularity	Layer boundaries	Variable
Virtualisation	Semantic based on layer roles	Flexible at the Interface type
Function re-use	Not obvious	More Intuitive

Models of distributed computation and coordination across functions



Models of distributed computation and coordination across functions

- Traditional system implementations of Internet stacks favour the *subroutines -model*
  - Functions execute in sequence across layers and every function completes its task before the next initiates (packet transmission/reception)
  - State is difficult to preserve across invocations of the same function
  - Challenge: introduce concurrency. Usually needs external scheduling (from OS) and synchronisation with queues
- Event systems favour the *event handlers model* (aka callbacks)
  - Functions can be called in any order by an event kernel and every function executes to completion
  - State is not easy but possible to preserve across invocations
  - Typically one service owns a set of handlers that share the same state
  - Lends nicely to cooperation through message passing communication

Models of distributed computation and coordination across functions

- Reactive systems use Co-routines
  - There is usually internal scheduling involved
  - Functions can sleep, pass control to another function and resume execution at any point while preserving state across calls

→ *Internet services are examples of distributed*
  - *reactive systems, or*
  - *event systems!*

→ *... where client and Server functions exchange state through message passing*

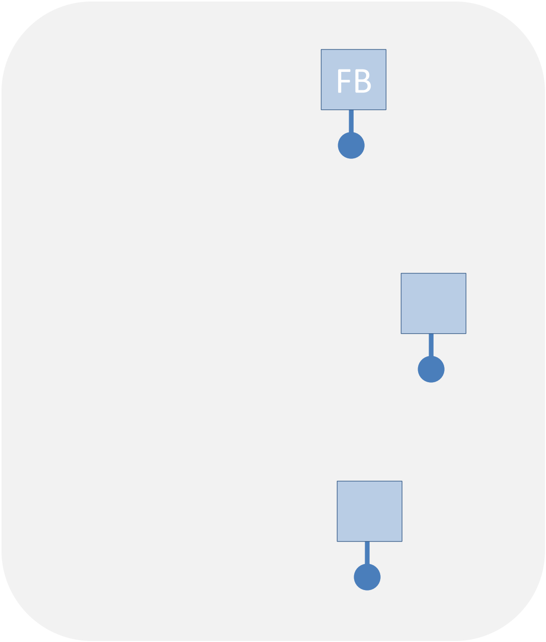
Autonomic Network Architecture

G. Bouabene, C. Jelger, C. Tschudin, S. Schmid, A. Keller, M. May, "The autonomic network architecture (ANA)“, Selected Areas in Communications, IEEE Journal on , vol.28, no.1, pp.4-14, January 2010

M. Sifalakis, A. Louca, G. Bouabene, M. Fry, A. Mauthe and D. Hutchison, "Functional composition in future networks". Journal of Computer Networks, Elsevier, Volume 55/Issue 4, pp. 987-998, March 2011

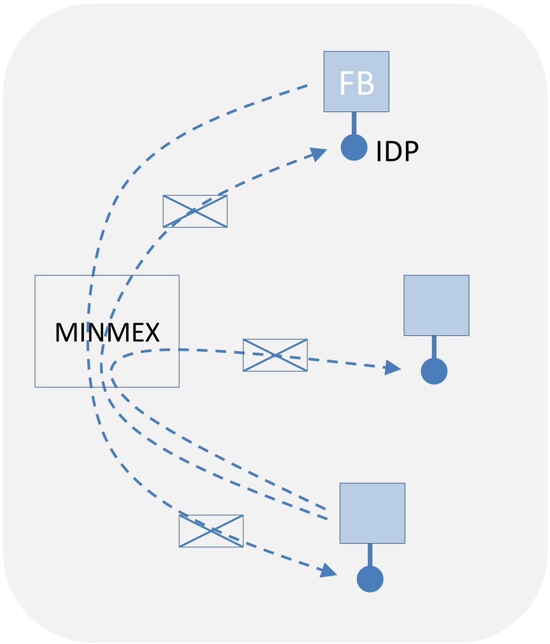
Foundations of ANA: Functional Block

- ANA provides a very similar level of functionality to OMNet++ (developed at similar time)
  - Albeit not a simulation environment but a virtual node software toolkit (think VM)
- Individual functions (FSM level) are provided by functional blocks (FBs)
- A pool of FBs is available within an ANA virtual node



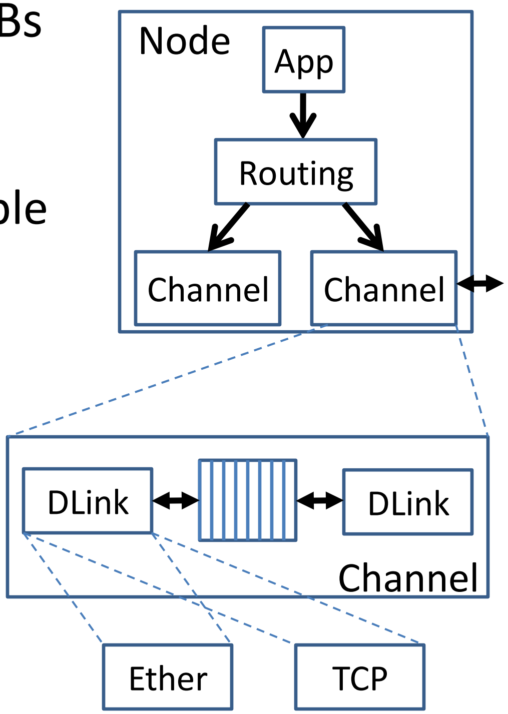
Foundations of ANA: MINMEX & IDPs

- FBs exchange messages via a message switching microkernel called MINMEX
- FBs receive messages at Indirection Points (IDPs)
  - The MINMEX switches messages to IDPs
  - IDPs do not belong to the FBs but to the MINMEX environment
  - One can think of IDPs as function pointers!
  - The association of an FB with an IDP can change at runtime
- IDPs in ANA enable
  - seamless and powerful runtime indirection
  - and re-direction (for functional composition)

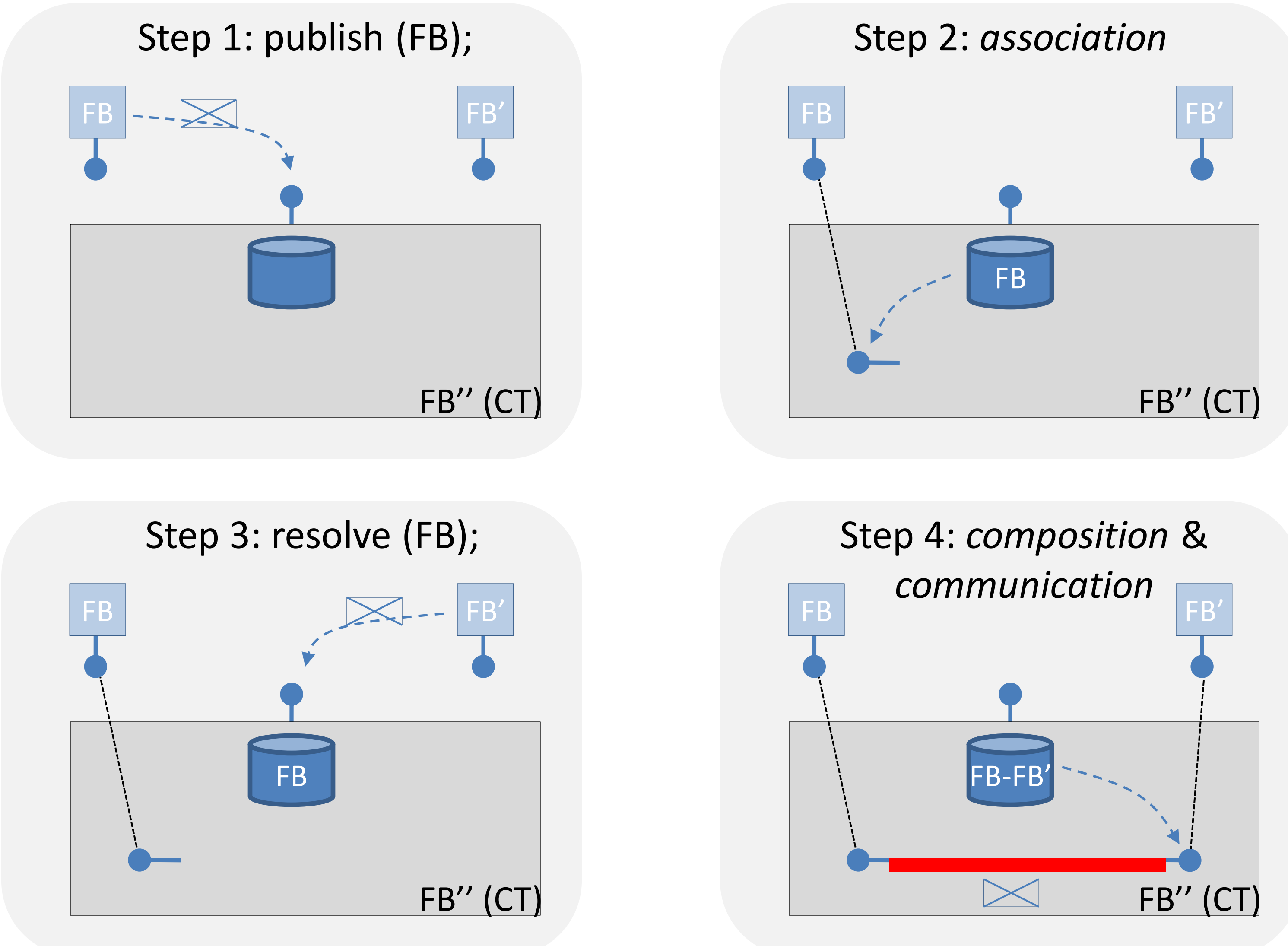


Foundations of ANA: Compartments

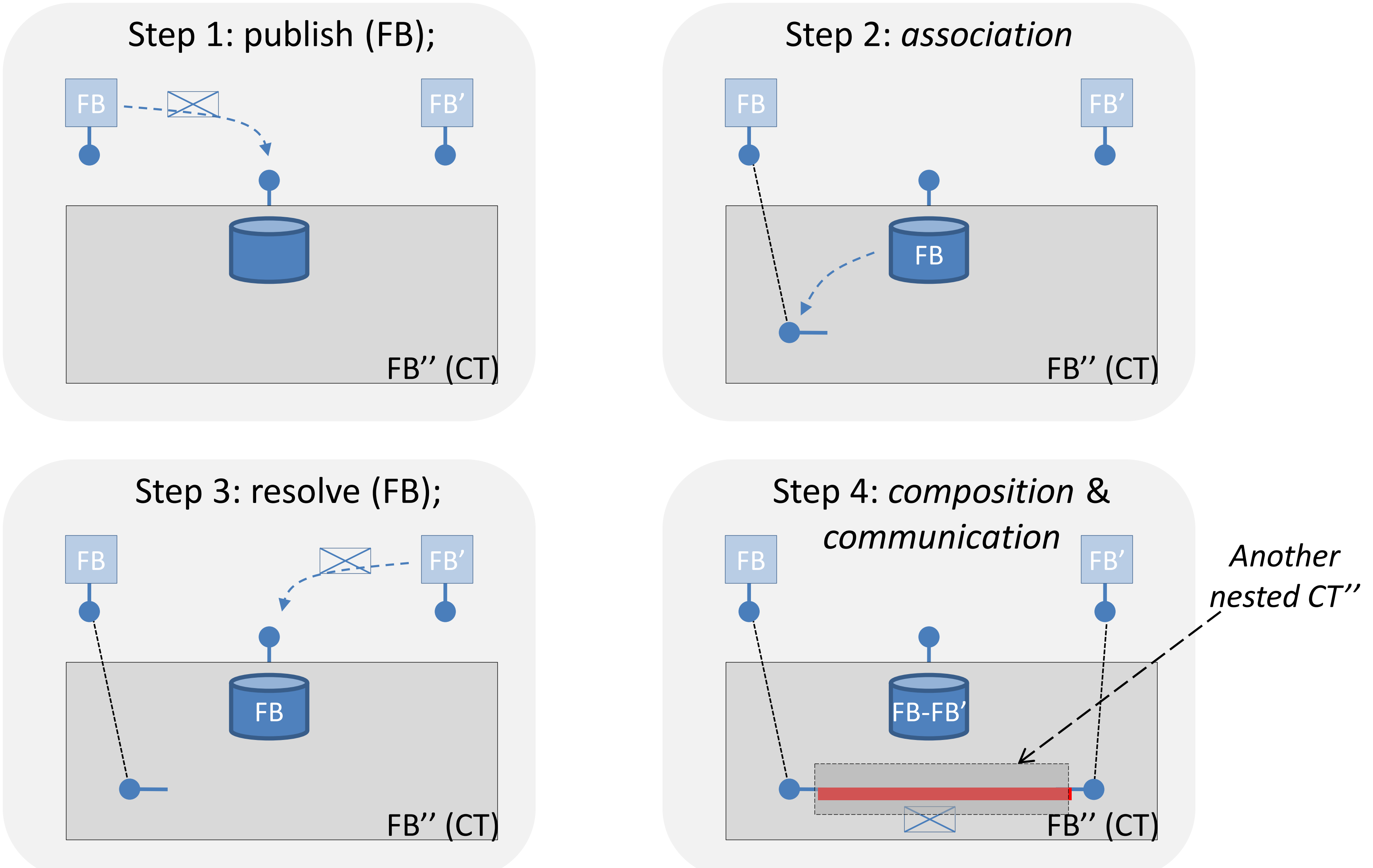
- Compartment (CT): service context implemented by FBs
  - Functional composition: FBs collaborate to provide complex service function
  - However, FBs do not belong to CTs: therefore re-usable for various services
- CT nesting: foundation for incremental abstraction
  - Network types
  - Layer functionality
  - Compound FBs
- CT functionality
  - A universal CT interface (IDP based – FB closure)
  - FBs can publish their services or discover existing services in CTs
- The notion of CT is not functional (*how to*) but semantic (*what*)



Foundations of ANA: CTs and FBs enable Information Channels



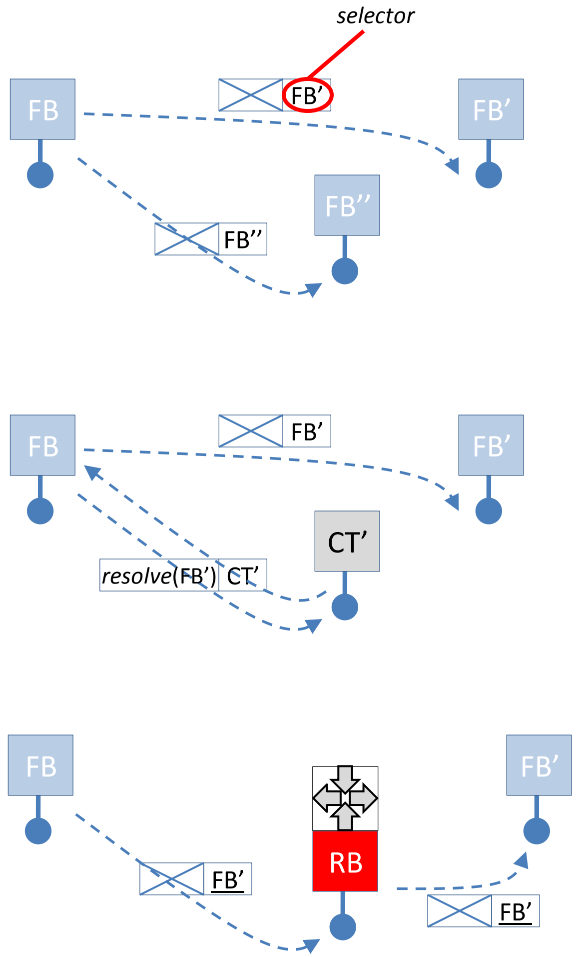
Foundations of ANA: CTs and FBs enable Information Channels



FBs: The network inside the node

- FBs can directly access other FBs through their IDPs (think of the IDP by analogy to a Netif)

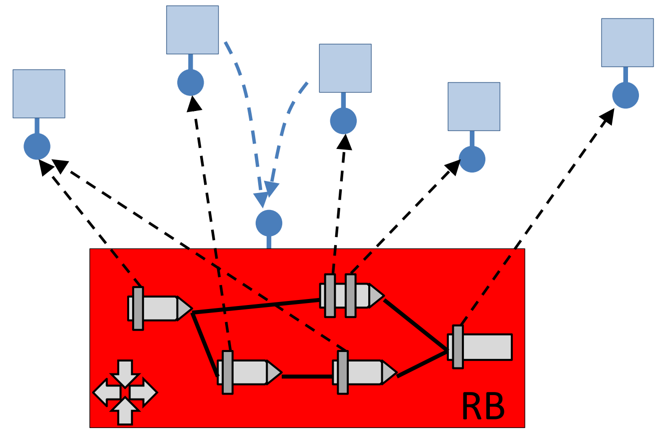
1. Manually
    - FB knows the ID(P) of other FBs
2. Discovery through Compartment resolution service
    - think ARP, DNS, other resolution systems
3. Agnostically, by sending to a ``router FB'' (RB),
    - which knows how to correctly deliver the message

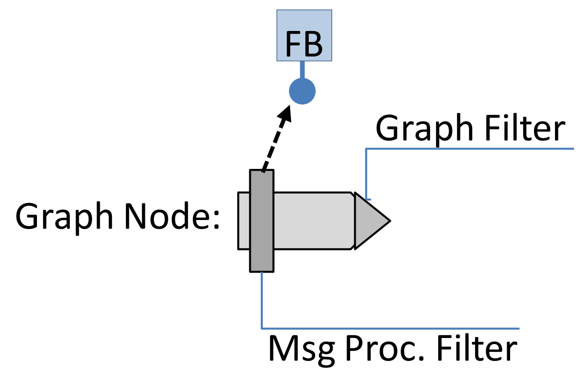


Functional Composition in ANA

Inside a Router FB

- Classification graph reflects how routing between FBs orchestrates a complex service
  - Nodes represent routing decision steps
  - Message Processing Filters (MPF) at each node may dispatch the message to an FB (if MATCH criteria met)
  - Graph Filters (GF) determine *flow-level* routing decisions (subsequent routing steps)
- When a message comes back, classification resumes from last routing step





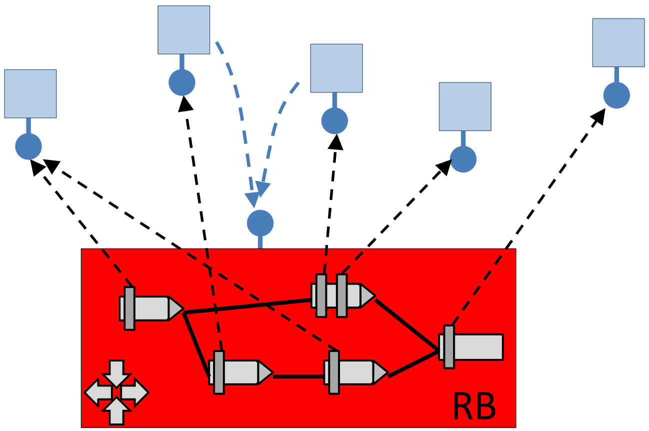
Functional Composition in ANA

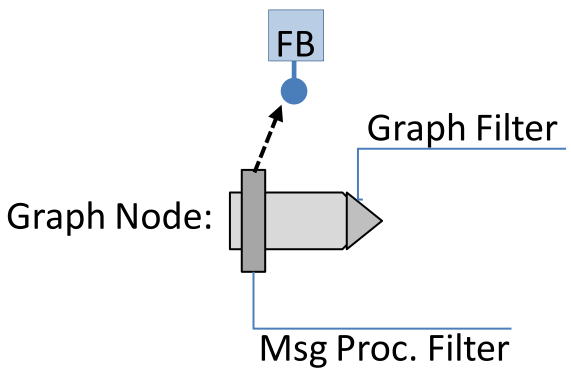
Inside a Router FB

- MPF and GF use filter rules to match bit patterns (protocol types, identifiers, state variables) in message, making routing decisions conditional

  A filter rule: *{<anchor> + <offset>, bit_pattern}*

- GF can switch ON/OFF parts of the graph based on external events



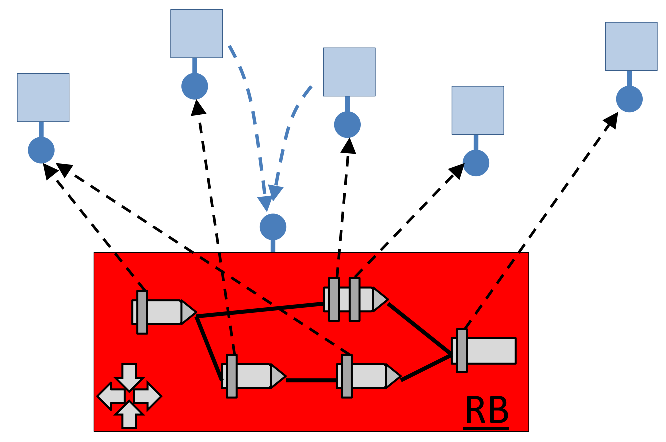


Functional Composition in ANA

Inside a Router FB

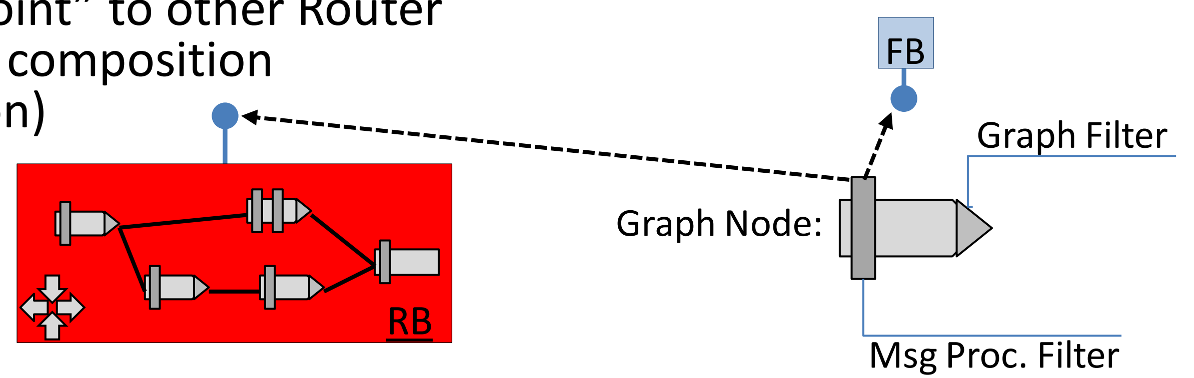
- MPF and GF use filter rules to match bit patterns (protocol types, identifiers, state variables) in message, making routing decisions conditional

  A filter rule: *{<anchor> + <offset>, bit_pattern}*



- GF can switch ON/OFF parts of the graph based on external events

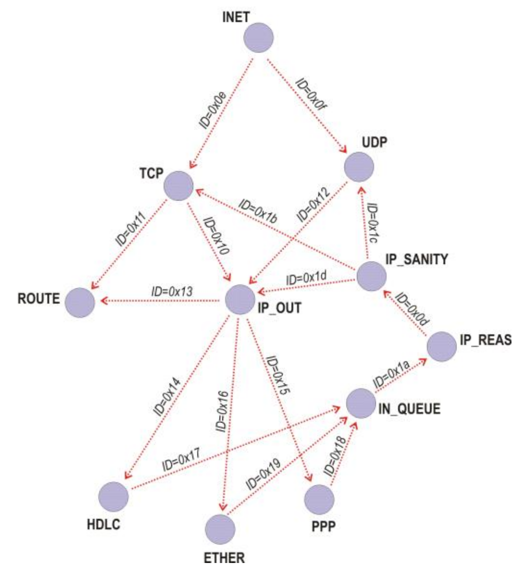
- MPFs may naturally “point” to other Router FBs allowing multilevel composition (incremental abstraction)

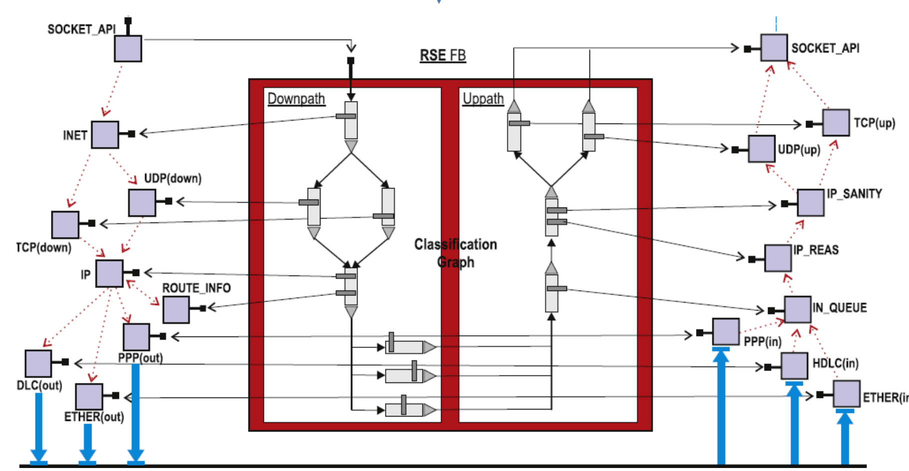


Functional Composition in ANA

- Service blueprints for Router FB
  - XML specifications describe FB relations
  - Parser generates the classification graph

```
<SERVICE>
  <F_REQS>      <!-- Functional Requirements -->
    <FUNCTION ID="0x01" TYPE1="AF_INET_API"> inet </FUNCTION>
    <FUNCTION ID="0x02" TYPE1="TRA_INOUT" TYPE1.1="SOCK_STREAM" TYPE2="AF_INET"> tcp </F
    <FUNCTION ID="0x03" TYPE1="TRA_INOUT" TYPE1.1="SOCK_DGRAM" TYPE2="AF_INET"> udp </FU
    <FUNCTION ID="0x04" TYPE1="NET_OUT" TYPE2="AF_INET"> ip_out </FUNCTION>
    <FUNCTION ID="0x05" TYPE1="RT_INFO" TYPE2="AF_INET"> route </FUNCTION>
        ...

    <SKELETON ID="0x0c" TYPE1='NET_IN' TYPE2='AF_INET'> ip_in
      <FUNCTION NAME="ip_reass" ID="0x0a" ORDER="1"/>
      <FUNCTION NAME="ip_sanity" ID="0x0b" ORDER="2"/>
    </SKELETON>
  </F_REQS>

  <E_REQS>      <!-- Event Requirements (optional) -->
    <!-- ### -->
    <INFO_USER ID='0x1d' NAME='router_enable'>
            ...
    </INFO_USER>
  </E_REQS>

  <R_REQS>      <!--Resource Requirements (optional) -->
    <INTERSCHED PRIORITY="NORMAL" />
    <QUEUE TYPE="OUT" SIZE="64K" DYNSIZE="NO" POLICY="FAIR"> output_queue </QUEUE>
    ...
  </R_REQS>

  <SRV_STRUCT>  <!-- Service Structure -->
    <!-- inet->tcp -->
    <FLOW ID="0x0e" FROM="0x01" TO="0x02">
            (foci{top}.tag == T_INPCB) &amp;&amp;
            (foci{top}.data->socket.so_type == SOCK_STREAM)
    </FLOW>
    <!-- inet->udp -->
    <FLOW ID="0x0f" FROM="0x01" TO="0x03">
            (foci{top}.tag == T_INPCB) &amp;&amp;
            (foci{top}.data->socket.so_type == SOCK_DGRAM)
    </FLOW>
        ...
  </SRV_STRUCT>

  <REG_INFO>    <!-- Service Registration Information -->
    <KEYWORDS> tcp_ip, network communication </KEYWORDS>
    <CONTEXT TYPE='local'/>
  </REG_INFO>
</SERVICE>
```



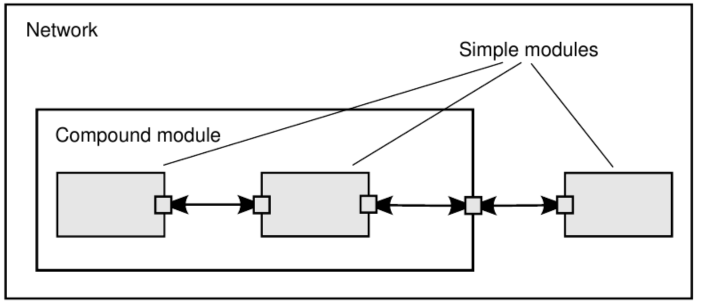


OMNet++

http://www.omnetpp.org/

OMNet++ Basics

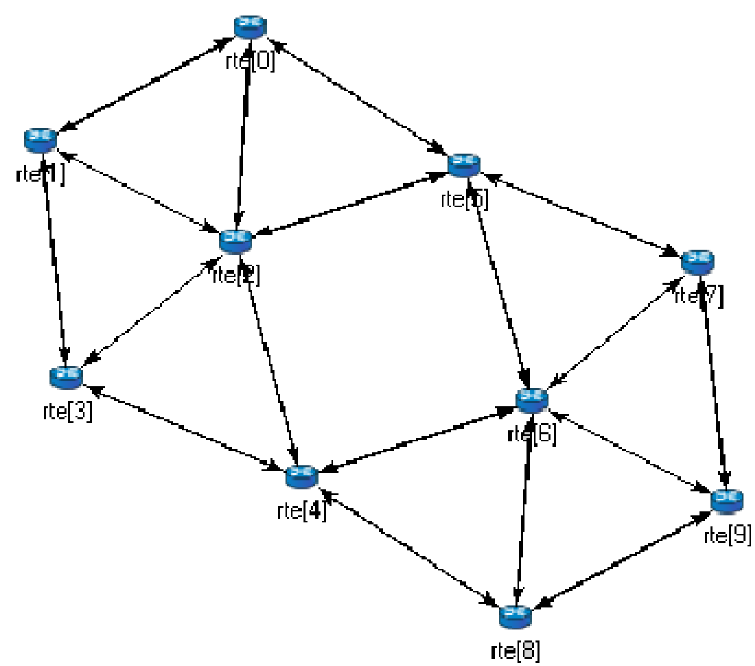
- Quite similar to ANA really!
- Modules (ANA:FB) communicate with message passing again
- *Simple* modules can be grouped into *Compound* modules that provide more complex functionality (service abstraction)
  - A network in OMNet++, is itself a compound module
- Modules exchange messages via *Gates* (ANA:IDP)
  - There are Input and Output Gates while IDPs are only Inputs
- … and along *Connections* (ANA:IC), but it is also possible to send them directly to the *Gates* of destination modules
  - An Input Gate and Output Gate can be linked by a Connection.



OMNet++ NED: Network Composition

Explicit Network Composition (ANA Compartments more dynamic)

```
network Network
{
 types:
  channel C extends ned.DatarateChannel  {
   datarate = 100Mbps; delay = 100us;  ber = 1e-10;
  }
 submodules:
  node1: Node;
  node2: Node;
  node3: Node;
  ...
 connections:
  node1.port++ <--> C <--> node2.port++;
  node2.port++ <--> C <--> node4.port++;
  node4.port++ <--> C <--> node6.port++;
  ...
}
```

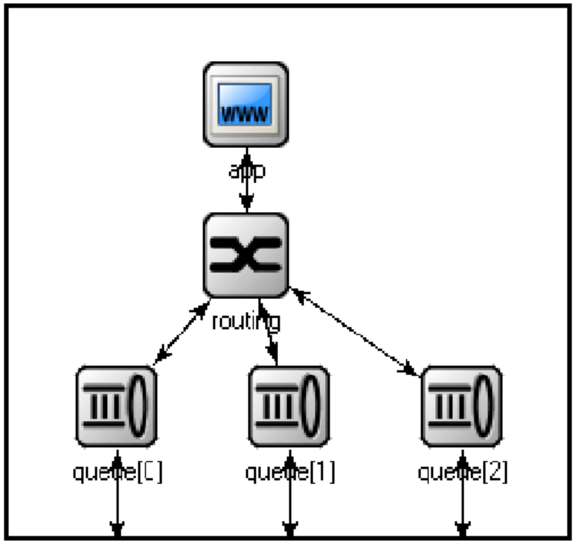


OMNet++ NED: Functional Composition

Functional Composition: NED language ≈ Service Blueprints in ANA

```
simple App {
  parameters:
    int destAddress;
gates:
    input in;
    output out;
}

simple Routing {  ... }

simple Queue {
  parameters:
    int capacity;
    @class(mylib::Queue);
gates:
    input in;
    output out;
}
```

```
module Node {
  parameters:
  gates:
    inout port[];
  submodules:
    app: App;
    routing: Routing;
    queue[sizeof(port)]: Queue;
  connections:
    routing.localOut --> app.in;
    routing.localIn <-- app.out;
    for i=0..sizeof(port)-1 {
      routing.out[i] --> queue[i].in;
      routing.in[i] <-- queue[i].out;
      queue[i].line <--> port[i];
    }
}
```



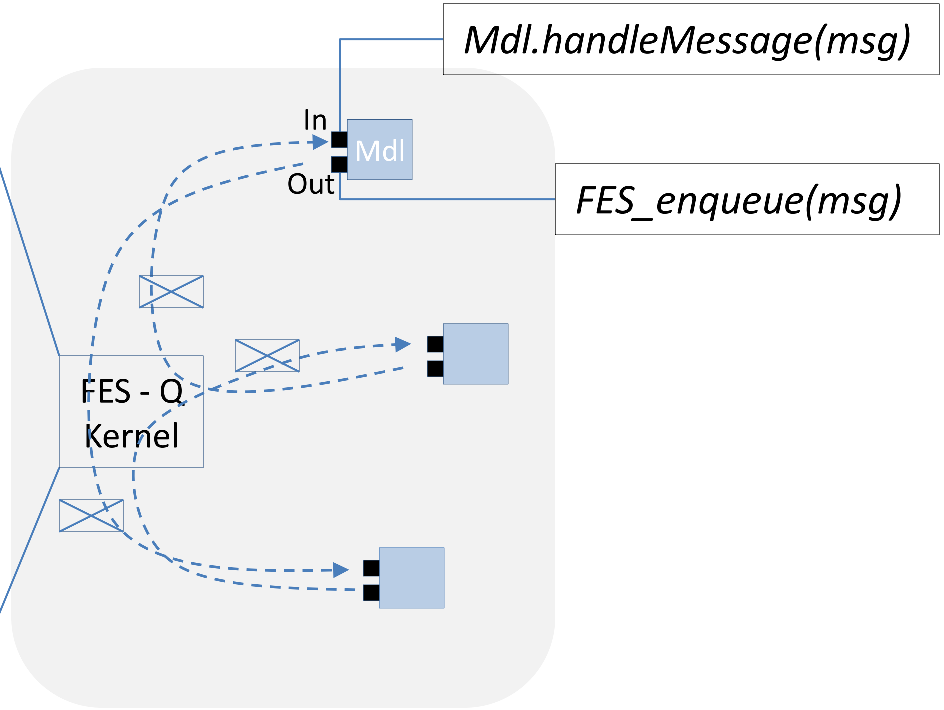
OMNet++ event kernel

- OMNet++ simulation event-kernel (ANA: MINMEX)
  - A FIFO queue (FES) holds timer and message transmission events
  - Modules register or delete events (message transmissions) at the FES every time a message is queued at an *outGate*
  - Event kernel “consumes” events from the queue by dispatching messages to recipient components (channels and modules).
    - Invoke module’s *handleMessage()* method, or
    - Send wake up signal to module’s *activate()* co-routine
  - Ordering of messages in the FES queue
    - Message with earlier arrival time (FIFO)
    - Message with smaller scheduling priority (user assigned)
    - Message scheduled first (EDF)

OMNet++ co-routine support

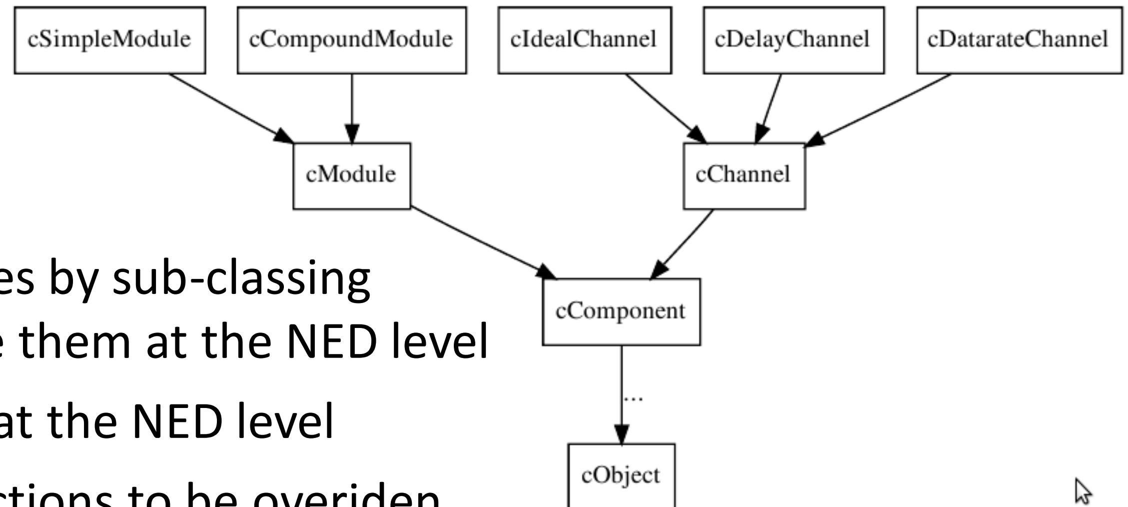
- Reactive-style cooperation between modules possible
- Possible with timers in ANA but not extended-support

```
init
while (FES not empty and simu not
complete)
{
    retrieve first event (msg) from FES
    t:= timestamp of this event
    m:= module containing this event
    if (m works with handleMessage())
       m->handleMessage( event )
    else // m works with activity()
       transferTo( m )
   // processing may insert new events
   // in FES or delete existing ones
}
finish simulation (write results, etc.)
```



OMNet++ NED and new modules (services) at compile time

- OMNett++ Class hierarchy



- Create simple module types by sub-classing *cSimpleModule* and define them at the NED level
- *cCompoundModules* only at the NED level
- *cComponent* member functions to be overiden
  - *initialize(), finish()* for initialization code and recording of summary statistics at the end
- Dynamic behavior of a simple module, override one the following
  - *handleMessage(cMessage *msg):* process/consumes message events, and returns
  - *activity()*: co-routine started at beginning of the simulation. Runs until end of simulation. Event kernel issues wake up calls to activate(). Msgs obtained with receive() from event kernel

OMNet++ runtime extension of services

Runtime composition

- dynamically create/destroy modules
  - for compound modules, all its submodules will be created recursively
- connect with other modules in the function composite
- Or use direct message sending (ignoring connections) with dynamically created modules
- E.g. when simulating a mobile network, you may create a new module whenever a new user enters the simulated area, and dispose of them when they leave the area

```
// find factory object
  cModuleType *mType =
    cModuleType::get("foo.nodes.WirelessNode");
// create (possibly compound) module
  cModule *m = mType->create("node", this);
  m->finalizeParameters();
  m->buildInside();
// create activation message
  m->scheduleStart(simTime());
// send direct message to new modules in Gate
  sendDirect (new cMessage("msg"), m, "in");
// create a bidir connection to another module
  m->gate("out")->connectTo(other->gate("in"));
  other->gate("out")->connectTo(m->gate("in"));
// disconnect from the other module
  m->gate("out")->disconnect();
  other->gate("out")->disconnect();
// detele the module
  m->callFinish();
  m->deleteModule();
```

ANA and OMNet++

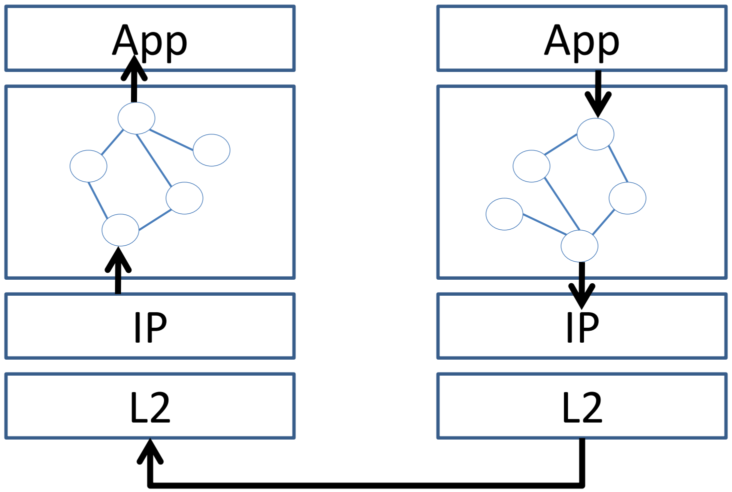
- They were developed at about the same time for different purposes
- ANA
  - EU project → research prototype
  - Cumbersome to program but runs on virtually any linux device (incl. Smartphones)
- OMNet++
  - Spin-off from the OPNET simulator → Semi-commercial product
  - Integrated programming environment
- Interfacing OMNet++ with ANA can be a challenging MSc project ;-)
  - Testing through simulation (OMNet++)
  - Automatic compile-deploy on an actual network (ANA)

Tau

R. Clayton and K. Calvert, "A Reactive Implementation of the Tau Composition Mechanism", Proceedings of the 1998 IEEE Conference on Open Architectures and Programming

Tau for evolvable transport protocols (remember RFC 1263 ?)

- Extensible transport-level services, and evolvable transport protocols
- Component-based backplane structure, into which protocol functions can be inserted or removed dynamically at run-time
- Assumptions about protocol functions in the environment are minimized
  - Protocol functions interact only through the Tau framework
- No explicit ordering on protocol processing,
  - Parallel implementations of protocol processing possible



Tau: Flow-based transport protocol composition

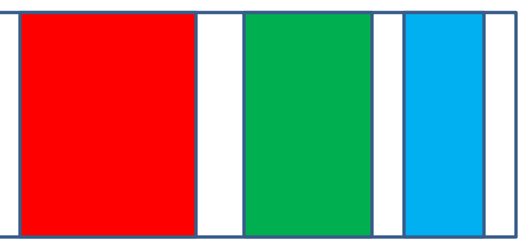
- Function composition determined by the communication source who decides what type of processing session data should be subjected to
  - e.g. auth, encryption, transcoding, etc
- Informs receiver with look-up table at a transport message meta-header
  - Header descriptors: the protocol function identifier as well as the size and location of the corresponding protocol header (if any)
  - Identifiers to common protocol header configurations (compound protocol functions)
  - Other general state

<u>Transport level message:</u>

- Red PDU is to be processed by prot. func_1

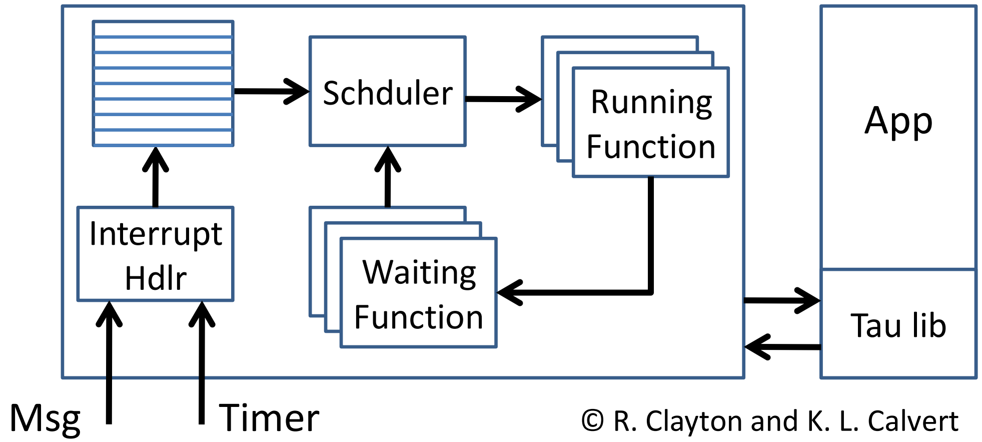
- Green PDU is to be processed by prot. func_2

PDU	PF ID
red	Pf_1
green	Pf_2



Tau Execution Architecture

- Signals (incoming messages) from the environment, along with any associated data, are captured with an Interrupt Handler (IH)
- IH queues signals on the Interval Queue
- Scheduler dequeues signals and pairs them with protocol functions waiting for the signals.
- For each pair found, the Scheduler starts executing the protocol function, passing the signal data, if any
- Pairing continues until all waiting protocol functions have been examined, at which point the current interval ends and the Scheduler goes back to the Interval Queue for the next signal



Tau runtime extension of services

- Similar to ANA and OMNet++: Event scheduler
- Generic Protocol Interface for protocol functions
  - Protocol functions must define message handling callback routines (message is the event)

    *typedef void **gpi** (void * **evnt**, void ***args**);*

  - Callbacks are then registered in the scheduler queue

    *handler-id **schedule-handler** (unsigned short **msglist**, gpi **pf**, void ***args**)*

    inserts the protocol function *pf* on the wait queue; returns an identifier that can be used in *cancel-mhandler()* to remove the protocol function from the wait queue

    *void **cancel-mhandler** (handler-id)*

  - *pf* needs to be manually re-registered as it will be removed from the wait queue when the scheduler encounters a message of the type indicated by the bit vector *msglist*

Tau parallelisation of protocol function execution loops

- *Show-stopper* functions: E.g. a sequencing function may determine that an incoming data unit is not in the receive window, and therefore should not be delivered to the user
  - Cause processing on a data unit to be aborted or delayed.
  - Parallel framework of Tau, other functions may process the data by the time a show-stopper is discovered.
  - Solution: segregate show-stoppers, and invoke them before any other processing
- Dependencies: E.g. fragmentation and reassembly, cannot be logically independent of other functions such as check-summing and sequencing
  - Solution: prescribe composition of such functions with others in the same layer, and permit recursion within Tau

Questions ?