Nodes & Functional Composition

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

This is something inside a single node. The idea is to organize the functionality inside the node such that is easier to adapt, so we can add/remove functions/protocols as we go/when we need to.

Omnet also somehow provides such a modular architecture. The way its functionality is organized could also be implemented in a non-simulated environment.

It's not that layers are bad, but the way the software is actually implemented doesn't necessarily need to follow this abstract concept all that strict. (it imposes penalties, like copying data time and again. The abstraction is nice when thinking about the functionality, though). The idea is to keep the semantic aspect, where the function belongs, but the code does not need to reflect that.

The problem with the layering is that there's a strict order. Introducing a virtual link that itself uses TCP (e.g. VPN tunnels), will give us huge problems, for instance when trying to load balance between the virtual link and the physical one...

Another example is packet switching over GSM. The GSM stack, which is a couple of layers itself, was basically treated as a link layer. Now, when running TCP over this, there's a problem: GSM has flow control, and so does TCP. The two flow control mechanisms actually competed against each other, leading to all sorts of problems.

Now, if we go to the world of components, things are more flexible for us. We can take any group of functions and define an abstraction layer, common interfaces for them. Each of the functions could have multiple interfaces, and serve different abstractions. For instance, TCP could have a data link interface, if we want to use it as a data link, and a transport interface, for when we use it as a transport. We can easily have loop detection in there, also we could use recursion :-D

Because we've seen in software engineering how nice modularity is, many came to the conclusion that this might be desirable for networking stacks, too.

3 models of coordinating computations that are distributed across some functions:

- Subroutines

\* execute functions in sequence, one calling the next, without preserving any state should one call back...

\* concurrency \*really\* challenging

- Event handlers (or callbacks)

\* Functions get called in any order by the event kernel

\* possible to preserve state across invocation (but difficult)

\* easily integrates with a message passing communication scheme

- Co-Routines

\* Functions cooperate, can pass control to other functions and resume at any point, all while preserving state across calls.

\* Could easily use message passing.

Traditionally, the subroutine model is used (very strict, think of each layer as a subroutine).

But actually, the whole networking is based on packet switching, which is basically message passing.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*Autonomic Network Architecture (ANA)\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Foundation:

Functional Blocks

\* each FB has the same interface

\* FBs exchange messages w/ each other

\* There's no differentiation between local and remote FBs

MINMEX (a message switching micro kernel):

\* MINMEX binds FB to IDP (indirection point) [idp is kind of like a network interface, through which FB can talk to each other - but they belong to MINMEX, not the FB]

\* Messages are sent from/to IDPs, thus the FBs behind the IDPs can be replaced at will (-> changing the binding)

\* allows flexible composition at runtime (example: TCP - put FB for flow control, reliable comms, error detection and flow control, i.e. 4 FBs. throw aways 3 of them, you basically get UDP)

Compartement (CT):

\* Abstraction for grouping FBs

\* CTs are a composition of FBs

\* FBs a CT is composed of are not owned by CT - they remain reusable

\* CTs are FBs themselves, too (this implies that CTs can be nested)

\* Compartements have a common interface

FB implements some function (says HOW TO DO SOMETHING), whereas CT represent WHAT IS DONE.

ANA kind of does (or at least allows) late binding as we know it from OOP..

Assuming we have two FB, and in some context they have to work together. (slide 18)

For that, they have to publish themselves (i.e. their services) to the compartement (this happens through message passing. It should also be noted that the compartement has a registry of some kind for this.). So the first FB publishes itself. By doing so, it associated. Now, the second one comes and wants to access some service (for this examples sake, the one the first FB registered). Now, the compartement associates the second FB, too, creating a channel between the two. A channel is basically a binding between two FBs, w/o the FBs knowing each other.

Now, the fun part, when the compartement is resolving the functionality the second FB wanted, it could resolve to another compartement, nesting other compartements and channels, even allowing recursion. This all happens at runtime, dynamically, it's not hardwired.

FBs have multiple ways to access other FBs:

\* "hardcoded": FB knows the IDP the FBs it needs to know

\* discovery via CT resolution service (like ARP, DNS, NetBIOS,...)

\* Agnostic via some "router FB" (RB) (which knows how&where to correctly deliver messages) [do we get atheistic and theistic methods, too? :p]

Intestines of a Router FB (RB)

\* two types of filters:

- message processing filters (MPF): pointers to IDPs. Match pattern in msg -> dispatch it to an FB, otherwise, off to graph filter which passes it on to the next RB

- graph filters (GF): connect nodes to each other (uses pattern matching for forward decision, too)

\* GF can switch parts of the graph on and/or off, based on external events

\* RBs are also FBs! (as the name already says), so we can use them as regular FBs, too!

Now, this composition can be done at runtime, automatically, based on some specification (using XML, for instance [mean editors comment: using XML here is a great way to make sure this will \*never\* get into the linux kernel - OTOH, if it is only needed by some usespace tool, that might be alright])

Nice fact: there is/was an implementation of this in the Linux Kernel, and it was running on desktops, laptops and mobile phones in a test phase.

\*\*\*\*\*\*\*\*\*

\*OMNet++\*

\*\*\*\*\*\*\*\*\*

- quite similar to ANA :D

- modules [ANA: FBs] communicate via msg passing

- modules can be grouped into compound modules

- message exchange via gates [ANA: IDP]

· difference to ANA: there are input and output gates, ANA knows just input for IDPs

- NED files for functional composition

· similar to ANA blueprints

- has a simulation event kernel (like ANA has MINMEX)

· FIFO queue

- OMNet++ also allows establishment of connections at runtime or sending msgs to gates directly (instead of sending through output gate, wherever that leads to)

\*\*\*\*\*

\*Tau\*

\*\*\*\*\*

[ to be done - its in ACS-13-1 ]