AAL – Greenfield Approach – Proposal - MRVL

Jerin Jacob

Agenda

- Requirements
- Green field architecture – Proposal
- API semantics walk through - Proposal
- Next steps and execution plan – Proposal

Requirements - 1

- Implementation details such as ‘Device’ and ‘Queue’ should not belong in this API specification
  - Abstracting it as *RAN objects* gives more flexibility to implementation without any downside
  - Making it as Device ID and Queue ID would call for one more indirection to access the data structures as we can not embedded all the info in an *int type*

- Have unified API semantics to abstract lookaside vs inline model
  - Hide the lookaside and inline complexity in implementation.
  - i.e Single application to work on both modes

- Focus on API specification and validation test suite
  - Leave Implementation complexity details to Implementor

Requirements - 2

- Provision to express all the combinations of profiles (current and future) in the AAL specification. We should have a capability to *link* different pieces/building-blocks together to form a new chain/profile for supporting all current and future DU/CU use cases.
  - Enable architecture to express the different ORAN splits like Spilt 6, Split 7.2
  - Have flexibility to define the input/output interface. Data provider interface can be CPU, PCI, Ethernet or another object in pipeline
- Coherent API model to support both CU and DU use cases
  - AAL must have a coherent architecture across DU and CU use cases
    - We should not define an entire spec looking at DU alone and leave CU for later
  - DU use case is a *fixed pipeline*
    - DU pipeline's RNF link(src and dest RNF) will be known at the time of profile creation
  - CU needs to be *dynamic pipeline based* on packet data such RRC, RLC protocol
    - CU pipeline's RNF link(src and dest RNF) will be known at the time of packet parsing

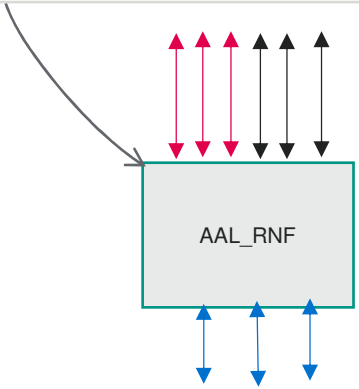
Requirements - 3

- *Vendor neutral* API specification – Ie. Keep only 3GPP specification related symbols in RAN object APIs

- Allow End-user to create new RAN function object and plugin to existing pipeline

Anatomy of a *Radio Access Network Function* object(RNF)

AAL RNF represents a fixed logical function in the 3GPP pipeline which can be implemented in HW or SW and as lookaside or inline or combination of all.



***Public* Base rnf APIs for ALL rnfs types**

- `aal_rnf_get_by_name()`
- `aal_rnf_stats()`
- `aal_rnf_link()`
- `aal_rnf_process_cb_register()`

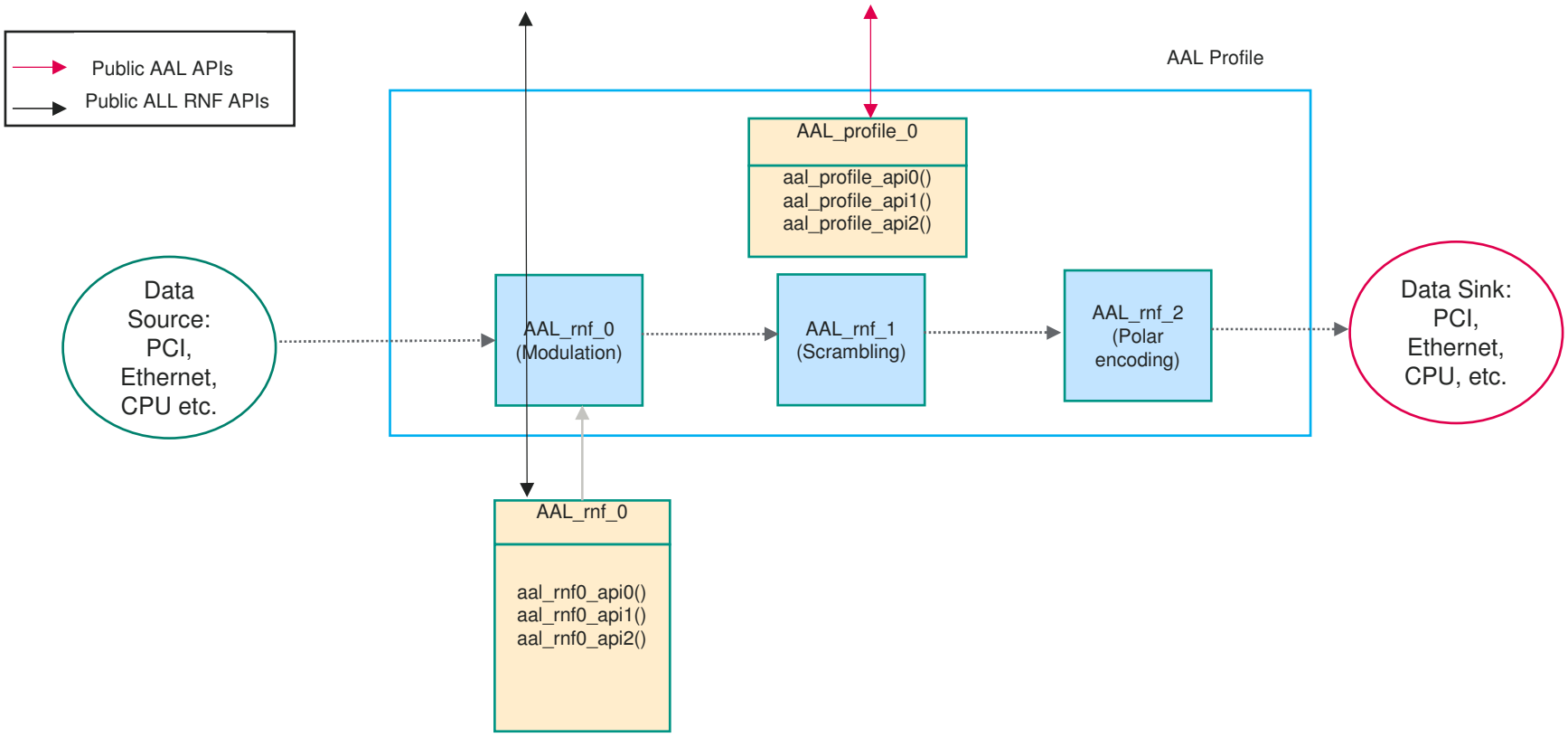
***Public* rnf specific APIs**

- `aal_polar_dec_from_rnf()`
- `aal_polar_dec_opearation1()`

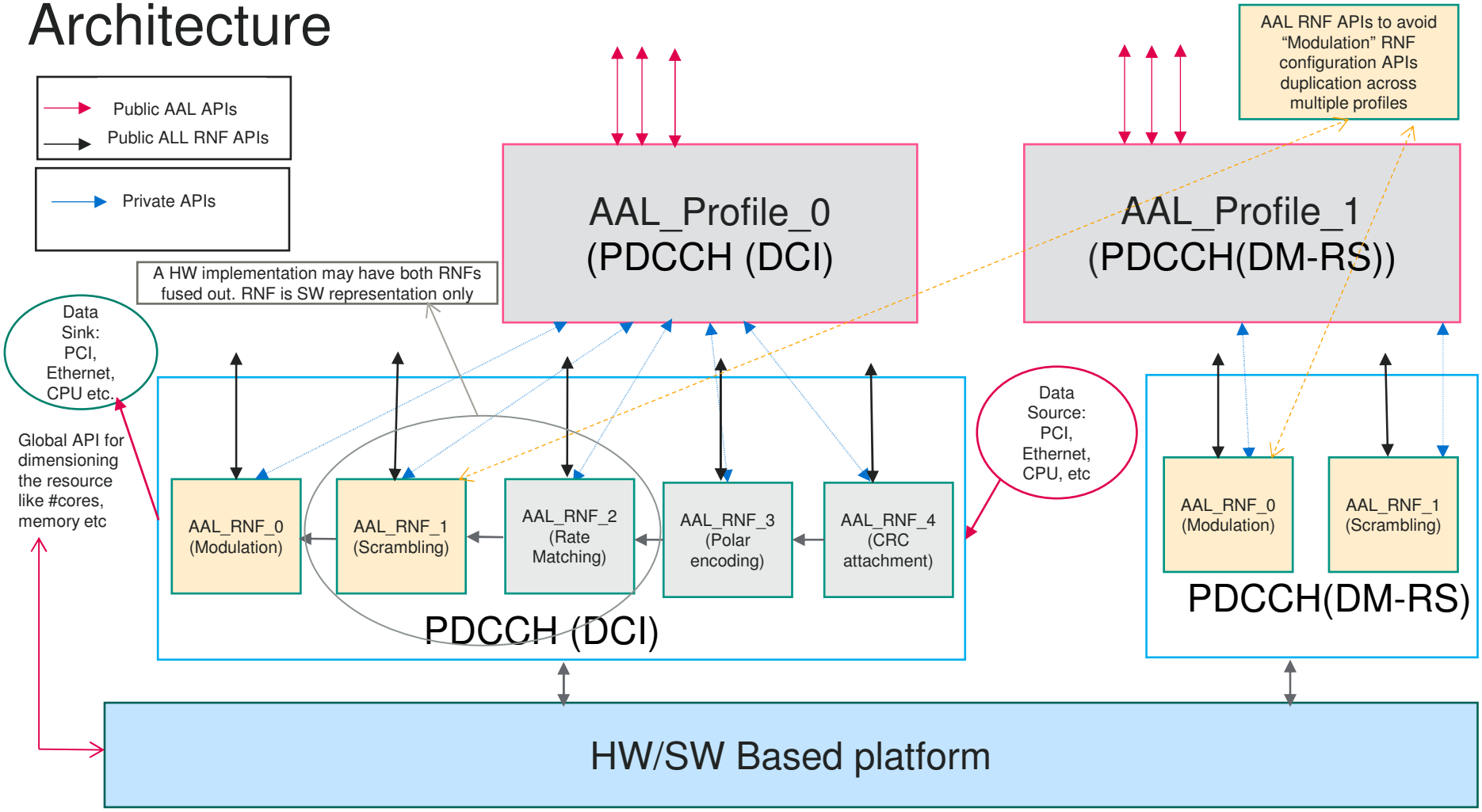
***Private* Vendor Specific Internal APIs**

- `mrvl_polar_dec_operation2()`
- `mrvl_polar_dec_operation3()`

Anatomy of an AAL Profile based on RNF objects



Architecture



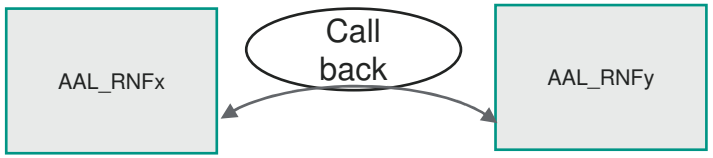
Architecture – Details - 1

- RNF API + Profile API split architecture is best as it caters to all type application requirements
- Future proof architecture
  - It would be possible to insert a customer-specific or application/deployment specific RNF in the pipeline without disrupting the rest of the profile
- RNF API concept can be used to reduce the configuration API duplication with multiple profiles
- Being able to visualize the architecture of AAL as functional units / processing stages (even though some functional units are always hard fused together) is very useful

Architecture – Details - 2

- RNF is a logical construct, not necessarily directly mapped to HW
- Purpose of the RNF is to reduce the "common logical" configuration API duplication.
  - For example, We don't introduce a sub level block, when a common block, like "modulation" needs to be configured then it need to duplicated in all the profiles API.
  - Instead, "modulation“(A RNF) specific configuration, we can reuse in all the profiles if we make "modulation" block as RNF in a profile.
- It gives application a logical view of the what are elements that consists of a profile.
- If an implementation does not support linking any arbitrary RNFs. The capabilities can limit to configure the RNF not the connecting them. Connection can be done only by “Profile” APIs. This would enable to use this architecture to work on RNFs that fused together.

Abstracting Lookaside vs Inline



- Each RNF supports a generic mechanism to register a callback when the rnf completes processing of an input bitstream

- When a RNF is running in Lookaside mode, implementation attaches a *default* callback handler to move bitstream from a self RNF to destination RNF.
  - *Default Handler* simply enqueues the output to the next RNF in the pipeline
  - The application will have the option to override the *Default Handler*

- For Inline mode, the hardware runs through the pipeline on its own i.e. no callback will be invoked by the *rnf*

- This way end Application stays the same, for both Inline and Lookaside modes

RNF Profile APIs – Structure and example prototypes

Type	API	Comments
Register	*aal_profile_register()*	Invoked by Implementation
Base Profile APIs (Valid for all profiles)	*aal_profile_for_each(void)*	Iterating over all the available profiles
	*struct aal_profile * aal_profile_by_name(const char *name)*	Get a profile handle
	*int aal_profile_stats(struct aal_profile * pf)*	Get profile stats
	*int aal_profile_start(struct aal_profile *pf)*	Start a profile
	*int aal_profile_stop(struct aal_profile *pf)*	Stop a profile
	*int aal_profile_data_source_set(struct aal_profile *pf, aal_data_src src)*	Set the data source(PCI, CPU, ethernet etc) for profile
	*int aal_profile_data_sink_set(struct aal_profile *pf, aal_data_sink sink)*	Set the data source(PCI, CPU, ethernet etc) for profile
	*int aal_profile_enq(struct aal_profile *pf, ..)*	Enqueue data to profile if data source is CPU
	*int aal_profile_deq(struct aal_profile *pf, ..)*	Dequeue data from profile if data sink is CPU
Profile Specific APIs (Example: PDSCH)	*struct aal_pdsch* aal_pdsch_from_profile(struct aal_profile *profile);*	Get PDSCH profile object from base profile handle
	int aal_pdsch_configure(struct aal_pdsch pdsch, struct aal_pdsch_cfg *)*	Configure PDSCH profile

RNF APIs – Structure and example prototypes

Type	API	Comments
Register	*aal_rnf_register()*	Invoked by Implementation
Base RNF APIs (Valid for all RNF objects)	*aal_rnf_for_each(profile)*	For iterating over all the rnfs in given profile
	*struct aal_rnf * aal_rnf_by_name(const char *name)*	Get RNF object
	*int aal_rnf_stats(struct aal_rnf *)*	Get RNF object Stats
	*int aal_rnf_link(struct aal_rnf *self , struct aal_rnf * dst)*	Link self to another RNF object
	*int aal_rnf_capa(struct aal_rnf *rnf, struct aal_rnf_capa *capa)*	Get RNF base object capabilities
	*int aal_rnf_process_cb_override(struct aal_rnf *rnf, aal_rnf_cb_t cb)*	Invoke the callback on process complete. Useful to abstract Lookaside vs. Inline. If source and destination RNF are operating in lookaside mode, the callback will be used to send the bit stream from source to destination RNF
RNF Specific APIs (Example: polar decoder)	*struct aal_polar_dec * aal_polar_dec_from_rnf(struct aal_rnf *rnf)*	Get polar decoder object from base RNF object
	*aal_polar_dec_opearation1(struct aal_polar_dec *)*	Do an operation on Polar decoder RNF object
	*aal_polar_dec_capa(aal_polar_dec *, struct aal_polar_dec_capa *capa)*	Return polar decoder capabilities

Pseudo code – From Application PoV

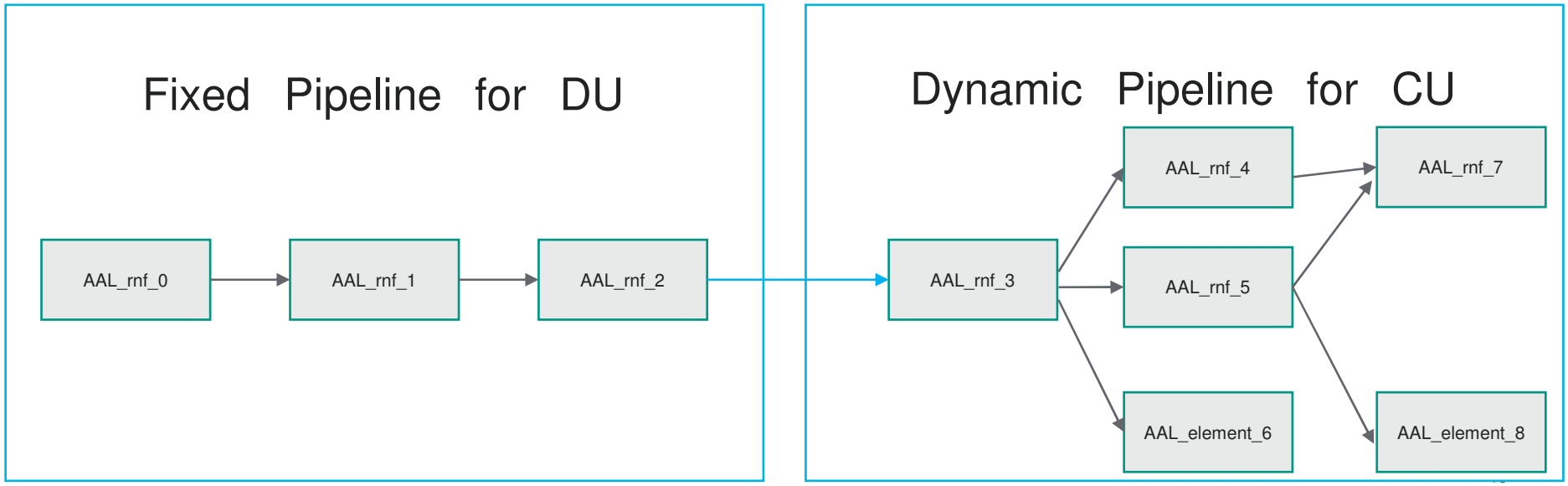
Step	Description
1	Implementation registers the supported RNFs using *aal_rnf_register()*
2	Implementation registers the supported profiles using *aal_profile_register()*
3	Application discovers the global resource(#cores, memory available) aspects using *aal_system_info()*
4	Application discovers the available profiles using *aal_profile_for_each()* iterator.
5	Application picks the profiles of interest by name, using *struct aal_profile * aal_profile_by_name(const char *name)*
6	Convert the base profile object to a specific profile object using *struct aal_pdsch* aal_pdsch_from_profile(struct aal_profile *profile)*
7	Do profile specific configuration, e.g. configure the *pdsch* profile using *aal_pdsch_configure().* Profile specific configuration creates and link the pipeline using registered RNFs in the implementation.

Pseudo code – From Application PoV

Step	Description
8	Get the RNF objects using *aal_rnf_for_each(profile)*
9	Configure/Override RNF object configure as needed (See *aal_rnf_* APIs)*
10	Set data source using *aal_profile_data_source_set()*
11	Set data sink using *aal_profile_data_sink_set()*
12	Start the profile *aal_profile_start()*
13	If the data source/sink is CPU, CPU calls *aal_profile_enq()* and *aal_profile_deq()* feed and get the data from profile. For example, If data source is *ethernet* and sink CPU, application needs to call only *aal_profile_deq()* to get the processed data from profile.

DU(L1) and CU(L2+) Integration

- DU(L1) use case does not need to inspect the packet, so it can be very light weight fixed pipeline where the source element and destination element are decided upfront before starting the profile
- CU(L2+) use case does need to inspect the packet so similar graph/pipeline line scheme can be enhanced to update dynamic pipeline based on packet content



Next Steps – Execution Plan - Proposal - 1

- Update GAnP for Requirements and Ground rules for Greenfield Approch
- Brainstorm and agree on the theme of the common API
- In order to accelerate the review and inline comments move to Gerrit or Github based contribution model
- Move the documentation to Doxygen (For API) and Sphinx(For user guide and architecture documents) and review in Gerrit or Github
- Start the API specification for common API *(Marvell can work on it)*
- Enable build infrastructure for docs and source code *(Marvell can work on it)*
- In parallel, start the RNF API specification(Can be done by multiple people)

Next Steps – Execution Plan - Proposal - 2

- MAINTAINERS Role
  - Whoever designs the initial header files, will be designated as MAINTAINER for the specific subsystem, Further enhancement should be Acked by the MAINTAINER for the specific subsystem
  - New MAINTAINERS can be added to the subsystem based on the contribution
  - Need to take lead of converging the technical discussions if there is conflict
  - Operate in vendor neutral way by keeping scope limited to 3GPP
  - Make sure to work on all the HW/SW implementation (i.e address *implementor* comments)

Next Steps – Execution Plan - Proposal - 3

- Criteria to accept PR (Pull Request)
  - Address the comments from all contributors
  - Acked by the one or more MAINTAINER(s)
  - When we add CI. Patch needs to pass the CI
  - Should we mandate the validation test suite for API ?
  - Should we mandate “Generic”(Not performance centric) implementation for the API? This will verify the API and act as test vehicle for test suite.


MARVELL™