SM.EXEC: A Network Service Logic Execution Environment Architecture

Anton Bondarenko
Moscow, Russia
anton.bondarenko@gmail.com
April 2024

***Abstract* — This article introduces SM.EXEC - an architecture of execution environment for network service layer applications. SM.EXEC is an abbreviation for finite State Machine EXECution environment.**

In short SM.EXEC - is an event-driven multi-threaded evironment for the finite state machines. SM.EXEC architecture is developed with network control plane of any type in mind - switching, routing, SDN, communication services, messaging etc. Compared to other FSM tool chains SM.EXEC assumes that SM-s are not hard-coded but rather are defined explicitly as a first-class system objects and managed dynamically by the system.

We explore how SM.EXEC is composed and how it can be applied to implement network control plane in the programmable manner.

SM.EXEC is the software architecture and it is developed with network elements build around general purpose CPU and OS, presumably - with these Linux/POSIX-based.

***Keywords* — network programmability, software defined networking, network service, execution environment**

I. Introduction

In this article we describe the architecture of the execution environment for network services which effectively exposes the network resources and supports extension and composition of earlier deployed services.

We will start with the short overview of the previous efforts in the programmable control plane and network application logic area in the next section.

In the following sections we will describe the design of the SM.EXEC architecture and the motivation behind the design solution made. And finally in the Conclusion we will get back to the impact and benefits different user groups may achieve by adoption of the proposed architecture.

II. Related Work

A. *Smart Home and IoT Platform Standardization Efforts*

The activities in this direction tightly connected to the two trends in the market of networked embedded platforms we mentioned briefly in the Introduction. The move towards Linux is closely related to the disaggregation of the platform resulting in the situation when microcontroller and SoC providers start to support third-party or acquired operating system - ARM Mbed (https://os.mbed.com/), FreeRTOS (https://freertos.org/), Silicon Labs Micrium OS (https://www.silabs.com/developers/micrium), SiFive Freedom E SDK (https://www.sifive.com/software), Zephyr OS (https://www.zephyrproject.org/) etc. The second trend is the adoption of IP for local and global connectivity.

Several middleware and API specifications utilizing these trends were proposed for home network NE programmability including RDK - the Reference Design Kit (https://rdkcentral.com/), Matter - ex-CHIP Alliance efforts (https://buildwithmatter.com/) and oneM2M (https://www.onem2m.org/). These specifications have different technical focus influenced by primary commercial interests of the industrial group behind each specification but these specifications have in common the coarse-grained components model which can become the limiting factors in some market scenarios if not addressed timely.

This component model assumes that relatively large components fulfill some high-level externally accessible functions which adsorb non-transparently all necessary resources and communicate over hard-coded APIs each of them exposes.

The important design goal of the SM.EXEC architecture and its лун differentiator is the avoidance of any hidden state and non-transparent usage of system resources including input-output, interrupts and multi-programming.

B. *Software-Defined Networking (SDN)*

The SDN [1] main concepts include disaggregation of the monolithic network switches into combination of hardware, OS and service logic application(s). The application portion also assumes exposure and centralization of the control plane over standardized protocols/APIs. This concept is highly tied to the network switch white-labeling.

SDN is mostly targeted at operator and data-center networks and therefore is limited in the application scope to the traffic forwarding and filtering. The decoupling of controller and data planes in SDN is important step towards the network programmability but in SDN world this concept assumes the physical separation of traffic forwarding function and control function by placing them in different boxes. This approach may not be necessary relevant to e.g. home network programming tasks. But the logical separation of these two planes does.

C. *Staged Event-Driven Architecture (SEDA)*

SEDA [2] is the architecture for highly concurrent web services proposed by Matt Welsh. This is another attempt to marry the thread and the event web server models.

The first model is using OS multi-threading mechanism to organize multitasking. This model (Fig. 1) is implemented e.g. by Apache Web Server [3].

The event-driven model represented on Fig. 2, also called the Event Loop is implemented in NGINX and Node.js and showed good performance [3].

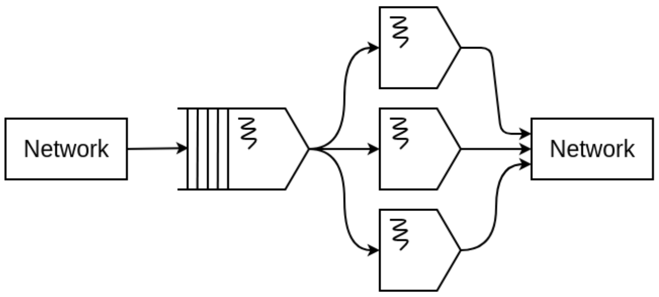

Fig. 1. Threaded server organization.

As it follows from the name SEDA is the event-driven architecture. SEDA-based server is a combination of several 'stages' each consisting of of event queue, thread pool processing events from stage queue, and stage controller. SEDA stages connects to each other by putting events to the next stage's queue (Fig. 3).

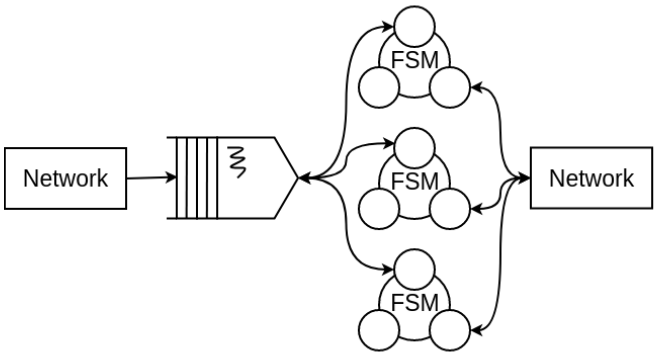

Fig. 2. Event-based server organization.

SEDA does the great deal of service load structuring by exposing explicitly load on different stages through the state of queues allowing to develop 'well conditioned services' [4]. But at the same time SEDA has some limitations including absence of support for sessions and unavoidable queuing between stages even in cases where uninterrupted execution of two consecutive stages by a single thread is preferable.

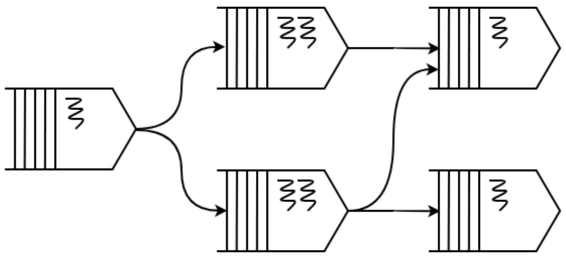

Fig. 3. SEDA sample stages configuration.

D. *Java APIs for Integrated Networks Service Logic Execution Environment (JAIN SLEE)*

JAIN SLEE [5] is a specification for an application server focused on telecommunications services. JAIN SLEE - is an event-driven architecture consisting of three main domains: the protocol Resource Adapters (RA), the SLEE itself and the Applications (Fig. 4). The main components of the SLEE are the input queue and the event router.

RA level is responsible for network protocol integration and provides wrapping of incoming protocol messages into the unified event object which then queued in the Event Router queue. One or several threads implementing event router logic then dequeue events and process them through subscribed application instances. Application in turn may fire events on the RA APIs.

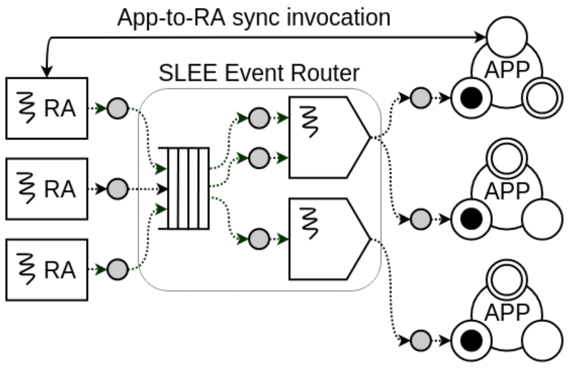

Fig. 4. JAIN SLEE sample configuration.

Application instances may persist between consecutive events within the same session been tied to the special object - Activity - representing the session i.e. the flow of events related to the same subscriber session.

In production deployments JAIN SLEE implementations showed very good performance results while remaining stable in multi-service and multi-protocol deployments. The service component model of JAIN SLEE inducts and supports highly structured application logic with explicit definition of service state machines. Protocol-agnostic is SLEE effectively conveying network protocol details wrapped in SLEE event objects between between RA and application layers.

JAIN SLEE specification was created with telecommunication applications in mind. It is easy to notice however that JAIN SLEE architecture can be efficiently applied to another domain like multiplayer online games, asset management systems or distributed industrial control systems. Not surprisingly it was proposed for SDN controller implementation [6]. Been implemented as an JAIN SLEE application SDN controller logic is just another communication service running in the JAIN SLEE container.

The JAIN SLEE utilize the model of Resource Adapters which used as an SLEE application interface to the external sources of the event - e.g. protocol agents etc. As a result the large components with hidden state and resources are still exist in the JAIN SLEE architecture. The sustainability of JAIN SLEE-based solution in presence of such components can be explained by the fact that most or even all protocols implemented by JAIN SLEE platforms are perfectly specified and very stable network protocols. With introduction of massive amount of custom APIs situation may turn to the worse quickly.

III. SM.EXEC Design Goals and Principles

A. *The High-Level View of the System in Scope*

Before we will start describing the proposed system architecture we need to answer some general questions: what the system shall do, what type of hardware will be used, what type of connectivity shall be supported.

The two latter questions can be answered based on the well recognized trends towards disaggregation and using IP as a transport protocol. So we will assume that some embedded platform using Linux or any embedded open RTOS communicating over IP will be used. This will mean among other things that potentially solution must be able to utilize multi-core microcontroller or microprocessor.

From the functional high-level perspective the design shall support running of network (and network - intensive) services allowing deployment and co-location of multiple applications developed by different supplier. Which implies in turn a requirement for an open programmable interface and exposure of the basic networking resources to the applications.

These two aspects - the multitasking and the network programmability we will consider in more details.

B. *Multitasking*

We already mentioned the events vs threads discussion which lasting for several decades. The main outcome of this discussion probably is that there is no single best system multitasking organization for any use-case. For web server design these concept seems to be on par if implemented properly. Therefore, we will move a little deeper and consider in more detail the characteristics of the tasks to be solved on the system being designed. And we will start from the the application characteristics analysis behind the JAIN SLEE architecture [7] and will extend it with what we learned since that time.

TABLE I. COMMUNICATION VS IT SERVICES

Service Domain Characteristic	**Communications**	**Enterprise/IT**
Primary task type	Reactive	Interactive
Primary invocation type	Asynchronous	Synchronous
Event granularity	Small size events, typically of signaling protocol message size (1-2kB).	Coarse-grained
Event rate, max per server	Up to hundreds of thousands	Much smaller for typical solution
Computation	Compute-intensive (network message processing)	I/O-intensive (disk and user interactions)
Availability requirements	Telecom gold standard 'five nines'	Less strict
Request processing time	Soft Real Time	Less strict
Preferred multitasking model	Event-driven	Thread-based or Event-driven

For communication systems and for systems which are close to them, e.g for cyber-physical systems, online games and industrial automation systems, the event-based model seems to be a clear winner. While business systems involve long-lasting transactions which can block the stack for a long time if implemented Event-Driven Architecture (EDA) small quickly handled signaling protocol messages and hardware interrupts provide the ready natural 'chunking' for the EDA approach.

Threading will still be important for any modern system of any range because even small embedded systems need some software means to utilize the multi-core capabilities of many modern microcontrollers.

Now let's check what category each home network service can be attributed to.

TABLE II. SERVICES IN THE SCOPE

Service / Service Type	**Communication**	**IT**
Forwarding IP traffic	yes	no
Filtering IP Traffic	yes	no
IoT Gateway	yes	no
Smart Home Controller / Gateway	yes	no
Convergent Communication Server	yes	no
ML/AI for traffic analysis	no	yes

Due to the differences in the characteristics of AI / ML applications, we can again return to the threads model and allocate some isolated compute resources for this task.

C. *Network Programmability*

There are many facets of the network programmability and the basic disaggregation into hardware, OS and applications layers is necessary but far not sufficient condition for appointing the network as programmable as we understand the network programmability. Some open API exposing network resources may still hide network agent state and the algorithm of system resources usage which may become a problem in complex multi-protocol and multi-service solutions.

We should require also to expose all aspects of system network logic starting from the socket level. In traditional system (and in all systems which we were discussing in this article) the components responsible for the network integration are part of the platform (Fig. 5).

But the desirable model may looks somewhat differently with the network integration components is implemented as regular applications deployed on the platform (Fig. 6).

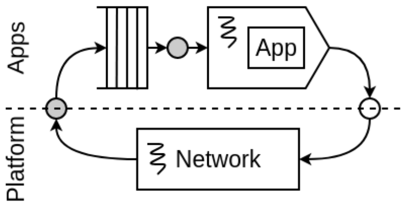

Fig. 5. The traditional design with network subsystem placed inside the platform

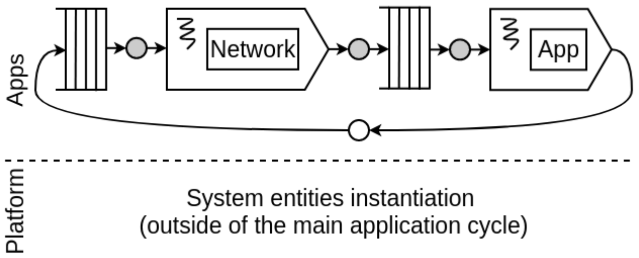
Apps
Network
App
Platform
System entities instantiation
(outside of the main application cycle)

Fig. 6. The network-as-an-application design

Another requirement we would like to add states that all network components should support reflection.

Reflection means that all network modules must allow to introspect traffic and protocol session states changes by third-party applications. This e.g. important for implementing such functions as checkpointing and migration of session, security monitoring and other.

From the practical perspective programmability means that programmable NE-s and their associated toolchains should be well suited for the agile, DevOps and CI/CD practices and be able to keep the system stable under frequent software updates. We will get back to this point below after description of SM.EXEC design.

D. *Design Goals Summary*

Let's recollect the main characteristics and features the designed network service execution environment (NSEE) and the associated toolchain should expose:

1) NSEE should work or should be easily portable to any common open disaggregated platform.

We mean here first of all Linux and its derivatives and any systems exposing same or similar application programming API.

2) NSEE must not limit developers in their choice of programming language(s) within the set of languages supporting compilation to the native code for the target platform.

To make the environment configurable we need to provide some means for dynamic deployment of event handling functions. This is achievable ether through implementation of the application execution environment supporting some type of Inversion-of-Control pattern or through supporting of the native dynamically deployable and linked libraries. The latter choice seems to be more natural for the world of the embedded solutions where the native programming using C and Assembler languages is predominant.

3) The network traffic handling logic such as routing and filtering should be decoupled from the networking hardware and OS APIs.

In the typical solution the core network functions are implemented immediately over OS API and doesn't lend themselves directly to the applications other than through configuration. With our purpose in mind to make the network functions a part of the application domain and to open doors for extension and deep integration of these functions with another applications we need to wrap the very first level of system API invocations into the NSEE applications. For example to extend the existing routing protocol logic with the awareness of the remaining charge of batteries of the potential intermediate nodes on the path the extension application must be provided with the access to the internals of the core routing application.

Another advantage of this approach is the removal of any possible otherwise barriers for the implementation and the usage of any protocols agents.

4) NSEE shall conform to the event-driven architecture (EDA).

This means that system is build as an reactive event processor and handles event objects which are the data structures wrapping the occurrences of the external and internal activities (flows of events or sessions). After been received each event is handled in a single threaded mode until the application state machine will reach the next state. Having done that system allocate the working thread to the top event in the incoming queue. We will describe the logic of the system in more details in the next section.

5) NSEE should support multithreading for utilizing the multi-core architectures and for efficient support on the same platform of mix of communication and IT services.

From the EDA perspective that means that multiple state machine instances related to the different external activities are served in a parallel threads.

6) NSEE services must expose the service state machines and allow to modify and combine state machines and services from different developers.

This means that each application must expose its internal state machine represented in some human-readable language. This

7) NSEE services must expose explicitly the current state of the application state machine.

Thus we decouple the data from the algorithms in our applications. This separation will simplify the application integration and extension and will allow development of the event handling programs in the clean functional programming paradigm and make them formally verifiable.

8) Platform operation and maintenance functional should be implemented as a regular NSEE services.

The 'everything-is-an-application' philosophy aims at making system easily extendable and flexible in any meaningful sense including operations and maintenance.

IV. SM.EXEC Design

In this part we will move on to the direct description of the SM.EXEC architecture. The name SM.EXEC the name is derived from an abbreviation for State Machine EXECutor and emphasizes the central role of the explicit state machine model in the proposed system design.

We will start with the general principles of SM.EXEC design, after that we will consider SM.EXEC main components and how they interact to implement the main processes in the system, and then we will look at some examples of system configuration. We will conclude this section by looking at the SM.EXEC architecture in the context of how it can coexist with the other programs in the same system and how the services executed by the NE-s can be migrated to SM.EXEC.

A. *SM.EXEC Principles*

1) All core SM.EXEC components are first-class program citizens.

We want to achieve the ability to make the very composition of the system including queues, threads, SM-s and other components reconfigurable and programmable. For this all the main entities of the system must be first-class citizens.

Robin Popplestone [8] gave the following definition of the first-class program objects:

a) All items can be the actual parameters of functions.

b) All items can be returned as results of functions.

c) All items can be the subject of assignment statements.

d) All items can be tested for equality.

We will add one more characteristics which will support SM.EXEC distributed deployments:

e) All items can be serialized and transferred to another host.

2) Performance over Hardware

The detailed analysis of poor performance reasons in reactive systems was present by Jeff Darcy in his article [9] where he describe the "Four Horsemen of Poor Performance":

a) Data copies.

b) Context switches.

c) Memory allocation.

d) Lock contention.

The idea then is if it is possible to avoid these biggest performance-killers for the most part of requests the designed server should perform well despite other possible design issues. This may result in less economical usage of RAM and other hardware resources but for headless network elements there is not too much expected side load and resource preallocation seems to be the optimal strategy. We're taking this approach as a starting point of SM.EXEC design.

3) Automata-Based Programming with Disaggregated States.

Automata-based (or SM-based) programming paradigm assumes the explicit definition of set of program states, the separation of transition functions from data (states), and the absence of side effects in transition functions (functional programming).

Strictly speaking finite state machine is the subset finite automata set and it is not Turing-complete. But by adding the randomly accessible memory to each state object we get the Turing-complete automata. So we use these terms interchangeably.

The application SM and its components become first-class citizens in such model. SM-based programming is highly relevant to the development reactive systems and has a number of additional advantages:

a) It produces self-documented services.

b) It allows the complete program validation.

c) It supports the structural integrity of the system under changes.

4) Everything is a Service.

In contrast to the traditional design of an application server as a stockpile of layers on top of hardware and OS, we place the developed NSEE at the center of our architecture and we integrate it with the platform using wrapper applications which are regular applications of NSEE.

Other features which are traditionally provided by the application server as part of the its builtin logic also become the regular applications. These features include fault management, load control, configuration management, security service etc.

B. *SM.EXEC Core Components*

We will construct SM.EXEC around the the concepts of Events, Queues and State Machines. Let's consider the core SM.EXEC components in more details:

1) Event Object

The event object encapsulates data about some internal or external occurrence e.g. the data field of the incoming network packet, the notification about the expired system timer, the interrupt from a peripheral device, etc.

Event objects, or just events where this will not lead to ambiguity, are pre-allocated structures stored in special 'home' queues - depots. So the dynamic memory allocation and data copying for the arrived events is avoided.

The data field stores information about the event that is passed between the calls of different state machine applications (see below). The memory areas in the event data field are allocated to different applications sequentially in the style of tape drive protocols and are not intended to be released until the event processing is completed in the system.

In addition to the data field events object structure includes the minimum set of necessary control fields. These fields include the pointer to the next event object in the queue, the size of the data field, the address of the depot for event object parking and some optional fields e.g. priority value, time stamp etc.

2) Event Queue

Queues are used for organizing sequential processing of asynchronously arriving events and for storing preallocated event objects pools.

Queue can be configured for synchronized or not synchronized mode depending on the number of provider and consumers users using the queue.

Queues in SM.EXEC can be bi-priority and multi-priority. The single-priority queues are not much more efficient than bi-priority so there is no need to implement them separately.

3) State Object

The state object stores the current state of the session (SM instance) through the lifetime of the session. The state object contains both the identifier of the current state in the SM and the extended session context. Compared to the event state object lifetime which is limited by the time the event is processed by the system, the state lifetime spans the whole session lifetime i.e. the state identifier will become one of the accepting states.

4) Applications (SM transition function)

Functions invoked for event processing on SM transitions are system native functions stored in dynamically loadable libraries. These functions have fixed signature consisting of the event and the state reference pair. The state object carry reference to the state machine description so it is also available within these functions scope.

In SM.EXEC themes such functions are called applications.

By accumulating all the context within event objects and state objects we assume these functions to be free of side effects other than changes to the event and state data fields. The event identifier is switched on SM transitions by SM executor (see below).

5) State Machine (SM)

SM is defined by a set of states identifiers, a set of event identifiers and a set of transitions corresponding to various pairs of an event identifier and a set identifier. Each transition can be associated with a call to one or more applications. Thus, for each pair of state and event identifiers, the SM object stores the identifier for the next state and a set of application references to be called upon transition.

The SM object is a singleton storing the SM structure, which is referenced by the state objects referring to this SM.

6) State Array (Depot)

To store used and unused state objects SM.EXEC provides an associative container which using the given search key either return the current active state object or allocates and registers a free state object for a new session.

7) SM Executor Object

SM executor is the wrapper object for thread-worker containing reference to the input queue. This object also holds the data field where the context of the queue and the related common data are stored. The lifetime of the executor data is the timeline of the executor itself.

8) Directory

System objects must be able to find each other during instantiation and during execution.

C. *SM.EXEC Core Libraries*

Here we provide the first set of libraries to be included in any SM.EXEC implementation and providing common functions potentially needed by most solutions.

1) SM.NET

A set of function for listening network socket and sending packets and datagrams and for wrapping payload into event objects.

2) SM.SOCK

A set of functions mimicking the socket API for easier porting of non-SM.EXEC applications over SM.NET layer.

3) SM.OAM

Fault, performance, configuration, security and logging function for the complete operations and maintenance support.

4) SM.HASH

Hashing functions available to all services for calculating hashing keys for storing state objects in arrays.

5) SM.JSON

This is the common set of (de)serialization functions for supporting SM.EXEC components mobility and for organizing of the exposure layer - API-s for the external solutions integration.

D. *SM.EXEC System Modules*

SM.EXEC also includes several service components that are not part of the operating cycle but are necessary for the efficient use of the system.

1) SM.LUA

SM.LUA is the Lua REPL (Read-Evaluate-Print-Loop) module which utilizes Lua language interpreter for optional system interactive management and configuration. SM.LUA is not an actual system component and system configuration may be .

2) SM.MEMORY

This is a common set of functions that together make up a complete cycle of simple memory management that can be used to manipulate data fields in event, state, and executor objects.

E. *SM.EXEC Operations*

1) Basic Operations

The minimal system configuration (see Fig. 7) includes a single execution thread belonging to the main application process, a single queue, a single executor object, and a single SM with a single state object associated with it.

The executor in an endless loop selects the next event from the queue and applies it to the current state configured by the SM.

All interaction of the system with the outside world, as we described, are wrapped into applications, which, among other things, are responsible for receiving and further processing external signals and for the preparing system's response.

2) Service Modifications

a) Changing System Components

As a first-class program citizens all SM-s, queues, Execs and other core system components can be added, removed or altered.

b) Changing System Configuration

The system configuration including allocation of queues to executors, assigning of state machines to states etc. is freely changeable programmatically i.e. from a regular application of SM.EXEC.

c) Checkpointing and Migration of SM Instances

SM.EXEC service migration is possible without interruption and without losing of the session data using the following sequence of actions: checkpoint all states, serialize all objects, relocate all system object to another installation of SM.EXEC, deserialize and instantiate all system object, apply necessary configuration on the target SM.EXEC installation, e.g. change appropriately service selection logic, reroute traffic to the new installation, remove all migrated system objects on the current installation.

d) State Machine Composition

In some cases it can be advantageous to provide a way for composing state machines instead of defining one joined state machine. As Matt Welsh noticed in [4] it would be preferable sometime to link stages directly instead of linking them through a queue.

F. *SM.EXEC Example Configurations*

In this section, we will look at examples of SM.EXEC configurations ranging from simplest ones to the emulation of previously discussed architectures in SM.EXEC.

1) The Minimal Configuration

This is the minimal SM.EXEC configuration which was described in the previous section.

All processing is done in the program main thread. If the input queue is empty the entire system is blocked freeing up system resources. But in the case of a single executor thread even if there is only one event object in the system such blocking is impossible because the event is loaded into the queue by the same thread that fetches it from the queue.

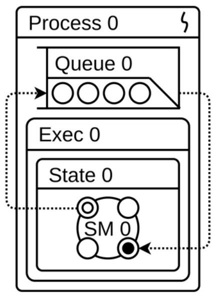

Fig. 7. The minimal SM.EXEC configuration

Blocking can occur during the execution of an application responsible for I/O or for socket operations and in that case the blocking is justified - if the system does not have event to process the system should be in a waiting state allowing redistribution of a thread in favor of another programs.

2) The minimal Configuration with an Extra Thread

If it is necessary to use an extra thread(s)an additional executor must be registered in the system assigned to the same input queue and configured with the same SM.

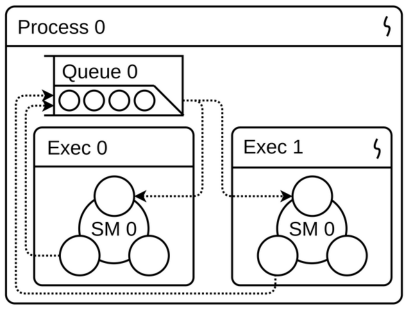

Fig. 8. The minimal SM.EXEC configuration with extra thread-worker

It should be noted that in this and in the previous examples, each executor does not assume a multiplicity of sessions: each SM is associated with a single state object (session context) which however is not a system limitation as we will see below.

3) *Multi-Threaded Configuration*

In this example the executor Exec 0 acting as a dispatcher receives external requests from the main queue, creates thread-workers and submits jobs to them. Intermediate queues 1..n can be thought of as simple buffers needed for negotiation between the dispatcher executor and the asynchronous worker threads.

After processing in the context of the dispatcher, the event is placed in the buffer (queue) of the executor allocated to it. After the event is processed in the worker executor, the event object is marked as free and returned back to the shared queue.

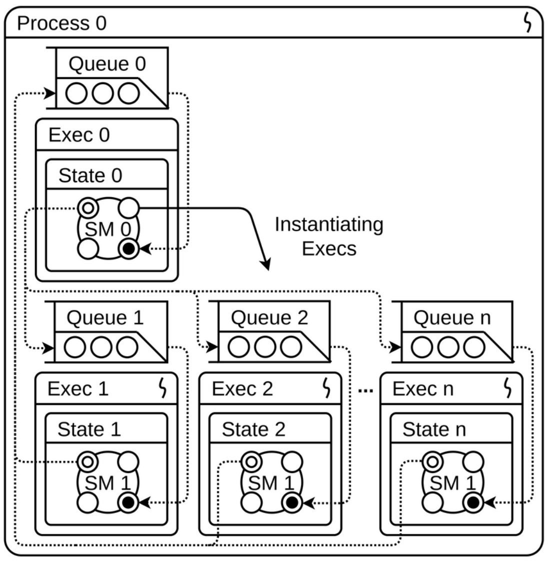

Fig. 9. SM.EXEC configuration replicating multithreaded server

Workers are freed after their SM-s reach their final state and can either be freed or returned to the pool (not shown in the figure).

We have given only one option, but variations are possible, for example, with a single intermediate queue. In the latter case, an imbalance in the load on worker workers is excluded, but loading events into the queue can create more locks.

4) *Event Loop Configuration*

The canonical implementation of the Event Loop pattern assumes the presence of an only thread and of multiple session contexts created upon arrival of each initial session events. The system contains a pool of pre-allocated state objects for storing session contexts. When the next event arrives at some unique session identifier a search is performed in the array of states (depot).

In the absence of a registered session context (the case of an initial event) the next free state object is assigned to the new session. For the second and subsequent events within the session the registered state object containing the previous session context is returned from the depot.

The executor Exec 0 acting as a dispatcher initiates request to the state array and fires event on the acquired state object and the SM referenced by the acquired state.

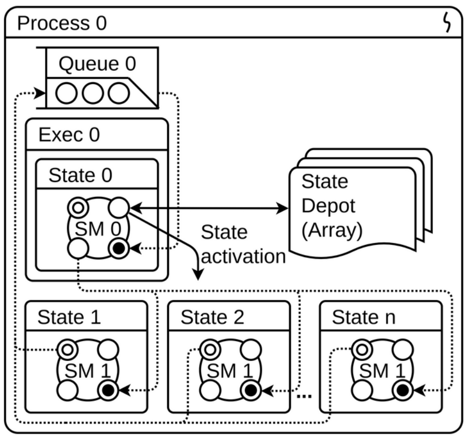

Fig. 10. SM.EXEC configuration implementing the Event Loop pattern

After being processed in the state context the event is returned to the queue and the state contexts depending on whether the state is accepting in the SM either freed or no. In any case the state object is returned to the state array (depot) after the event processing.

5) *JAIN SLEE*

Fig. 11. presents a simplified version of the configuration of a some JAIN SLEE server. The figure shows three levels of architecture: Resource Adapters - SLEE (Event Router) - Applications.

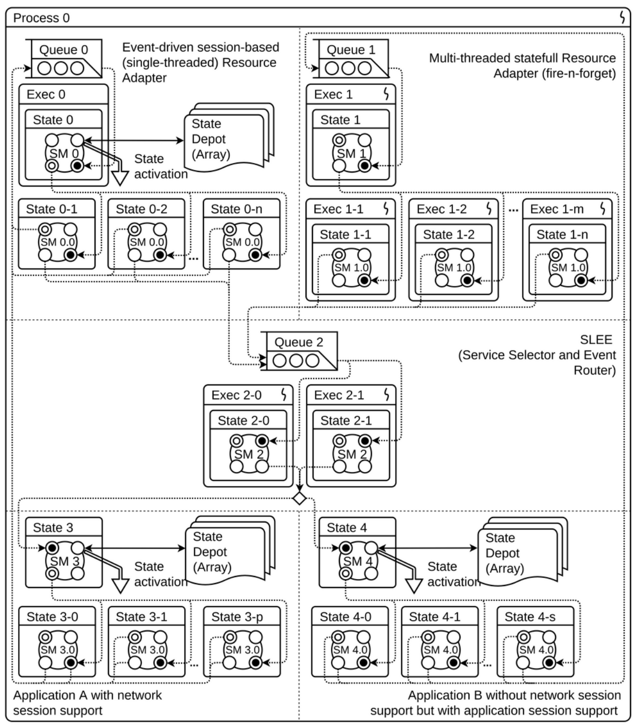

Fig. 11. SM.EXEC configuration implementing configuration on Fig.4.

6) *SEDA*

In a more general example on Fig. 12. the SM.EXEC configuration repeats on the high level the SEDA configuration example shown in Fig. 3. The rotated by 45 degrees square sign in the event streams connection points denotes a queue selection point.

RA on the left attached to the Queue 0 is stateful and its dispatcher checks for the state object for each incoming event in the state array and passes event to the extracted state object. RA on the right side of the picture (attached to the Queue 1) is stateless.

Events produced by each adapter can be processed by its own assigned application and for stateless RA it is possible to invoke both applications sequentially.

G. *SM.EXEC Gradual Migration and Coexistence with Other Applications*

In this section we will show how SM.EXEC can be gradually implanted into the NE and coexist with legacy components and how SM.EXEC-based services can interact with other services. Ideally, all services will be migrated to SM.EXEC and interaction will be reduced to SM composition but in the early stages at least we should assume that some services will still represent closed monolithic components.

As we discussed earlier we assume that the system under modification is the disaggregated network element running under Linux or similar OS and communicating over IP.

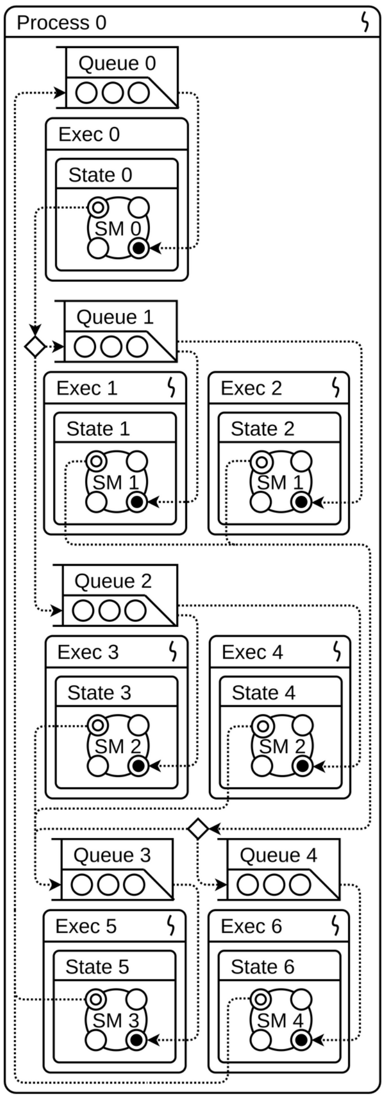

Fig. 12. SM.EXEC configuration implementing sample on Fig. 3.

On Fig.13. some possible generalized phases of migration towards SM.EXEC are presented:

1) *Adding SM.EXEC Services to non-SM.EXEC NE*

The figure shows the simplest initial use case for an SM.EXEC solution with an existing NE application. Legacy 3rd-party applications do not interact with SM.EXEC applications.

2) *Adding Traffic Sniffer to SM.EXEC*

The figure represents the SM.EXEC libpcap client with which the SM.EXEC services access the traffic of non-SM.EXEC services in an observer mode.

3) *NE Logic Migrated to SM.NET/SM.SOCK*

In this figure SM.EXEC (SM.NET) network applications completely controls network resources and exposes the network API towards both legacy non-SM.EXEC services and towards services implemented on SM.EXEC. In this case SM.EXEC applications can not only observe the network element traffic but also control it. The SM.SOCK module implements an API similar to UNIX / Linux network sockets and allows to minimize the effort of adapting 3rd-party applications.

4) *The Clean SM.EXEC*

NE 3rd-party logic is ported to SM.EXEC and NE FSM is fully exposed to extension and modification together with new applications developed on SM.EXEC.

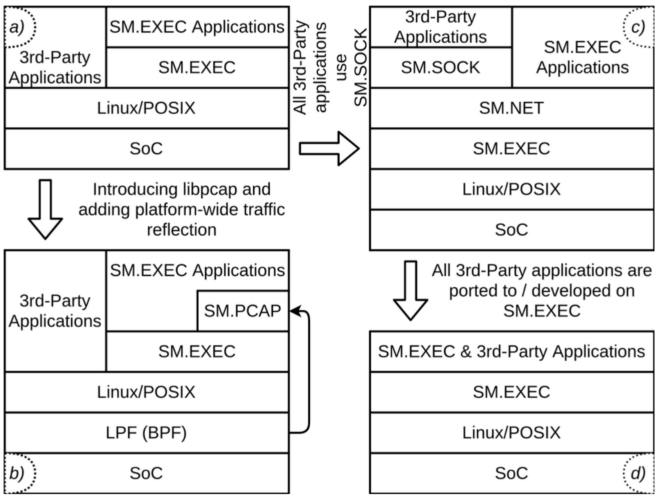

Fig. 13. SM.EXEC possible migration paths

H. *SM.EXEC Sample Applications*

In this subsection we are providing the list of sample network functions which can be implemented on SM.EXEC. This list is not complete and intended to be used as illustration of the possible scope of the solutions which can be potentially built on SM.EXEC implementation.

We intentionally providing both applications related to home networks and applications for other market segments to better highlight the actual scope of SM.EXEC.

1) *Functions deployed in home network*

- Wi-Fi Router
- 6LoWPAN Border Router
- Firewall
- Traffic Anomaly Detector
- Home PBX
- Smart Home Controller
- IoT Controller
- MQTT broker
- Parental control

2) *Home network related functions deployed in operator network*

- SDN Controller (flow controller)
- Traffic Anomaly Detector

- Class 5 Switch / feature server
- Smart Home application cloud platform
- IoT application cloud platform
- Dual subscription solutions

3) Operator b2C services

- IMS Application Server
- Charging Triggering Function
- IM/SSF
- Home Zone charging solution
- Dual subscription solutions
- Parental control

4) Operator B2B services

- Voice VPN (vPBX)
- Traffic routing logic for the compliance needs (MiFID II etc.)
- Office Zone charging solution
- IoT application platform / M2M solutions

5) Operator services - Network

- 5G AF
- 4G, 5G base station control/signalling portion
- Session Border Controller
- IMS x-CSCF
- Signaling Agents (DRA, DEA, 5G SCP)
- CAMEL gsmSCF
- Traffic anomaly detector
- Fraud prevention solutions

V. Conclusion

In this paper we presented the initial "paper-based" prototype of the new type of service execution environment which was designed to meet all the requirements we formulated in the beginning of the paper.

The main differentiating characteristic of the proposed architecture is the extended disaggregation of the software system components which will let to remove all blockers on the path towards free product and service composition and the complete Network-as-a-Service (HNaaS) proposal.

Along with total components decoupling through the proposed architecture introduces a quite strict structure which is intended to keep the subscriber and service provider networks safe, efficient and sustainable under the increasing flow of new user services and home network elements. This creates also a solid foundation for the successful and efficient implementation of Agile, DevOps and CI/CD software development practices.

On Fig.14. the main design features of SM.EXEC and the associated advantages are presented briefly.

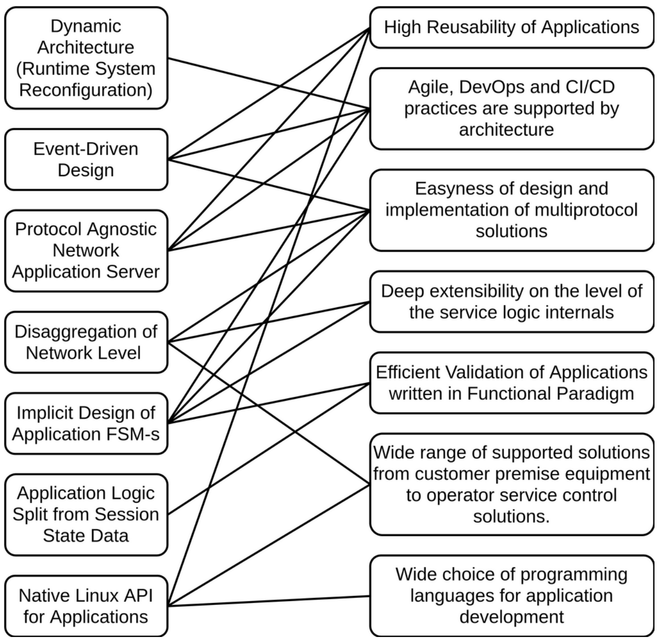
Dynamic Architecture (Runtime System Reconfiguration)
Event-Driven Design
Protocol Agnostic Network Application Server
Disaggregation of Network Level
Implicit Design of Application FSM-s
Application Logic Split from Session State Data
Native Linux API for Applications
High Reusability of Applications
Agile, DevOps and CI/CD practices are supported by architecture
Easyness of design and implementation of multiprotocol solutions
Deep extensibility on the level of the service logic internals
Efficient Validation of Applications written in Functional Paradigm
Wide range of supported solutions from customer premise equipment to operator service control solutions.
Wide choice of programming languages for application development

Fig. 14. SM.EXEC features and the related advantages

In this research we were focusing on home networks and the related service but we have to notice that SM.EXEC architecture development was initiated with wider scope in mind including mobile service provider service level engines, SDN controllers, large-scale agent-based simulation research systems etc. With regards to home networks this means that SM.EXEC can be also used on the operators' side for implementation of the control and monitoring solutions necessary for efficient implementation of the NaaS model.

Having the same application development platform both on the service provider side and on the subscriber side will make application development much more efficient and will give good foundation for application code reuse.

The proposed NSEE architecture is optimized for services which are extensively using and built are around of various network protocols. But it can be used for any services which are close to the network services in the sense of usage of system and network resources.

References

[1] Open Network Foundation, SDN Architecture ONF TR-502, 2014, [Online], Available: https://opennetworking.org/wp-content/uploads/2013/02/TR_SDN_ARCH_1.0_06062014.pdf

[2] Matt Welsh, David Culler, and Eric Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services", Proc. of SOSP 2001, Lake Louis, Banff, Canada, 2001.

[3] Ryan Dahl, "node.js", jsconf09, 2009, [Online], Available: https://tinyclouds.org/jsconf.pdf

[4] Matt Welsh, "A Retrospective on SEDA", 2010, [Online], Available: https://matt-welsh.blogspot.com/2010/07/retrospective-on-seda.html.

[5] JSR 240: JAIN SLEE (JSLEE) v1.1, 2005. [Online]. Available: https://jcp.org/en/jsr/detail?id=240.

[6] Caio Ferreira et al, "Towards a Carrier Grade SDN Controller: Integrating OpenFlow With Telecom Services", The Tenth Advanced International Conference on Telecommunications (AICT), Paris, 2014.

[7] David Ferry et al., "JAIN SLEE Tutorial. Serving the Developer Community", 2003. [Online]. Available: https://www.oracle.com/technetwork/java/jain-slee-tutorial-150035.pdf

[8] R. J. Popplestone: The Design Philosophy of POP-2. in: D. Michie: Machine Intelligence 3, Edinburgh at the University Press, 1968

[9] Jeff Darcy, Server Design, [Online], Available: http://pl.atyp.us/content/tech/servers.html, 2008

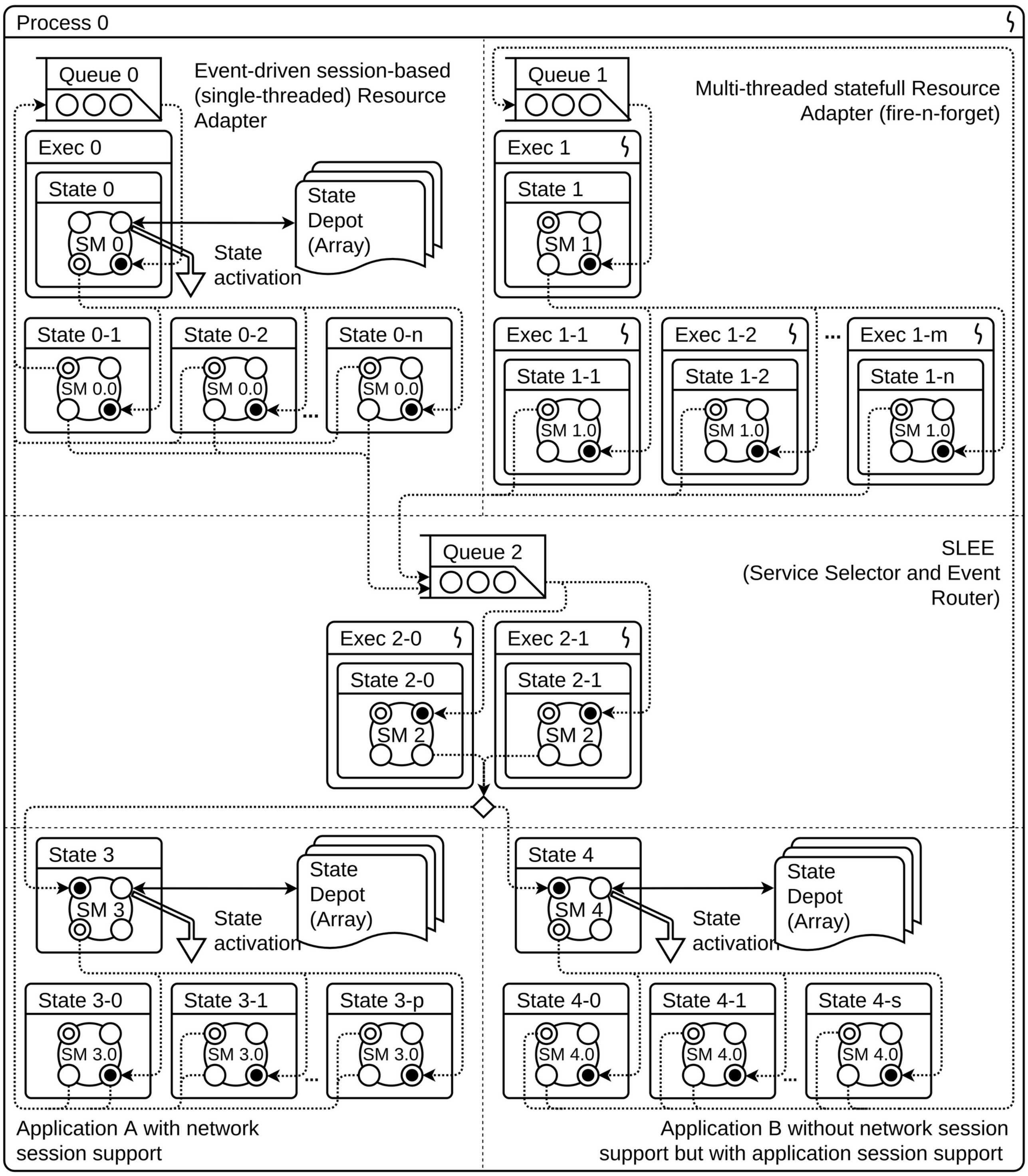

Fig. 15. Enlarged copy of Fig.11