A Survey of Dynamically Adaptable Protocol Stacks

Vangelis Gazis, Member, IEEE, Eleni Patouni, Graduate Student Member, IEEE, Nancy Alonistioti and Lazaros Merakos

Abstract—The continuous development of new networking standards over the last decade has resulted in an unprecedented proliferation of interfacing technologies and their associated protocol stacks. Never before was such a wide gamut of network architectures, protocol configurations and deployment options available to network designers. Alas, this significant increase in flexibility has come at the cost of an increased complexity in network management tasks, particularly with regard to the accommodation of performance requirements. Especially in mobile settings, this is due to the greater probability of unforeseen communication contexts that renders the efficient provisioning of multiple dissimilar protocol stacks a challenging task. To address this unpredictability, several approaches based on the dynamic adaptation of protocol stacks during runtime have been proposed and investigated over the years. This article surveys major research efforts dealing with the introduction of a dynamic adaptation capacity into protocol stack subsystems. To this end, we present the respective architectures with a focus on their functional entities and their particular mode of operation. Most importantly, we elaborate on the various design approaches to adaptability and the entailed degree of coupling between protocol stack—and layer—entities and their impact on resource allocation models. Furthermore, we classify these research efforts according to a taxonomy for non-monolithic protocol stacks and discuss design trade-offs inherent in each class. We conclude the article with a summary of the key design principles for adaptable protocol stack architectures.

Index Terms—Dynamic adaptation, next generation networks, protocol stacks, reconfiguration, autonomic communications

I. INTRODUCTION

Over the last decade, several wireless access systems, such as cellular systems (e.g., GSM, GPRS/GERAN, EDGE UMTS, cdma2000, etc), broadband WLAN type systems (e.g., IEEE 802.11a/b/g), fixed wireless access systems (e.g., IEEE 802.16d/e), broadcast systems (e.g., DAB, DVB-T/S/C/H, DMB) and short-range wireless systems (e.g., Bluetooth, IEEE 802.15.3a, IEEE 802.15.4) became available while new ones are being developed rapidly (e.g., 3GPP HSPA). Similar developments characterize landline and carrier networking, where several innovative technologies (e.g., Gigabit Ethernet, WDM/DWDM, ADSL/VDSL, Ethernet over SDH, IP over SONET, MPLS/GMPLS, etc) have emerged and/or are being intensively developed.

The ongoing proliferation of networking standards is reflected in standardization activity; over the last decade, the IETF has proposed and standardized more new protocols than in the previous two decades. Revisions to existing protocols add to the intensification of standardization activity and the same holds true for other standardization groups (e.g., 3GPP). These ever faster standardization cycles promote a ‘mix-and-match’ approach to the definition of protocol stack standards (e.g., 3GPP frequently reuses IETF protocols in its mobile network architecture). Not surprisingly, the increase in the number of supported protocols, aggravated by the definition of multiple releases and versions, has introduced significant complexity in network management tasks. It is estimated that the network infrastructure of a typical medium-size ISP features more than 1,000 different protocols—a daunting management challenge.

In light of this proliferation of networking technologies and protocol stack standards, the traditional monolithic (or ‘silos’) approach to protocol stack design often proves inefficient. In a monolithic design, an amount of functional duplication (e.g., segmentation and reassembly, integrity checks) is inevitable in multiple protocol stacks. In current practice, all the protocol stacks of a network system operate continuously and in isolation to each other. However, as the number of standards to be supported by a communication subsystem increase, the possibility that some of its enclosed protocol layers are actually idle at a given time increases rapidly. This is due to existing standards for protocol stacks, most of which have not been designed under multi-system and multi-protocol assumptions. However, idle protocol layers unnecessarily consume valuable system resources (e.g., processing cycles, data buffers, etc) which, in mobile devices are far from abundant. The latter claim a continuously increasing portion of Internet-capable end user systems, thus multiplying the aggregate impact of resource waste. Given that the observed proliferation of wireless access systems is likely to persist and that future networks will have to support multiple standards and protocol stacks, adaptation has emerged as a key technological enabler of multi-standard protocol stack systems.

The dynamic modification of the protocol subsystem in operating network elements, which extends from the dynamic modification of protocol parameters to the runtime incorporation of additional protocol layers and protocol stacks, is among the primary objectives of ongoing reconfiguration research [1]. Protocol stack reconfiguration is the dynamic adaptation of the protocol stack runtime system and its enclosed elements in a manner which does not compromise its ability to provide services. It is now commonly understood that dynamically adaptable protocol stacks will be a fundamental technological en-
able of next generation networks where user mobility [2] and application portability across different network systems render a highly unpredictable communication context and, thus, limit the set of a priori assumptions that can be safely made about it. This unpredictability has motivated several research efforts on customizable and adaptable protocol stack frameworks. Equipped with artificial intelligence features (e.g., machine learning), next generation networks will leverage adaptation capacities to autonomously adjust their protocols’ mode of operation and thus provide optimal performance in all possible contexts [3], [4].

The rest of the paper is organized as follows: Section II sets the stage for adaptable protocol stacks by presenting their various design approaches and unique objectives. Section III presents the prominent software frameworks for protocol stacks and classifies them from the viewpoint of software engineering (since the adaptation capacity is realized at the software level). Section IV presents the class of composable protocols stacks and elaborates on their design blueprints while Section V does the same for so-called customizable protocol stacks. Section VI draws on the prior presentation to discuss salient features of adaptable protocol stacks and to highlight their design tradeoffs with regard to system design, complexity of application, composition model, flexibility of configuration, architecture openness and performance. Finally, Section VII concludes the paper with key design principles to consider in engineering dynamically adaptable protocol stacks that strike a balance between flexibility and performance.

II. PROTOCOL STACK TECHNOLOGY: DESIGN CHOICES AND CLASSIFICATION

All adaptable protocol stacks have particular design principles in common presented subsequently. For purposes of a more spherical presentation, herein we classify adaptable protocol stacks from two different viewpoints: a) the software engineering viewpoint concerned with the organization of the protocol stack’s software (i.e., the "how"), and, b) the viewpoint concerned with the objective that protocol stack adaptation strives to achieve (i.e., the "why").

A. Design principles

The dynamic adaptation of a protocol stack basically results from its associated software architecture and the amount of flexibility it supports [5]. Because software architecture defines the organization and structure of software and its constituent elements, it also determines the boundaries of its feasible adaptations. Depending on the particular organization of its software architecture, protocol stack adaptations may affect either parts of or the entire protocol stack. The execution environment of the protocol stack (i.e., the communication subsystem’s kernel) and its software architecture affects—and, to a significant degree, determines—whether protocol stack adaptation can take place at bootstrap-time and/or during run-time. Adaptations of a protocol stack that involve the replacement of operational (i.e., running) protocol code may induce a disruption in the (otherwise continuous) operation of the affected protocol layers. Depending on the extent of desired changes to the protocol stack, the latter’s support for change management operations upon the software code of individual protocol layers and the level of control upon inter-protocol layer communication, this disruption may propagate throughout the protocol stack, thus effectively enforcing a total reboot of the communication subsystem.

It is apparent that the dynamic adaptation of the protocol stack subsystem imposes significant engineering challenges that cannot be efficiently addressed by the conventional so-called ‘silo’ approach to protocol stack design. The latter typically suffers from extreme vertical integration between software and hardware and lack of horizontal integration across protocol stacks and their associated communication standards. Historically, this architectural rigidity has been conventionally considered as the price to pay for an optimized performance. However, modern design approaches and formal software engineering technologies support a wide gamut of adaptation capacities whilst achieving comparable performance [6].

The gain in flexibility facilitated by adaptable protocol stacks does not come at the expense of protocol interoperability. As subsequent sections detail, several engineering approaches to adaptable protocol stack frameworks have been developed over the years. Considering that each dynamic protocol stack framework is based on a unique architecture, interoperability between different frameworks is inexistential. However, this does not mean that peer protocols implemented using different software frameworks are not interoperable. It is compliance to a protocol’s formal specification (e.g., state machine) that determines the interoperability of protocol implementations, not the software artifacts supporting protocol development.

B. Protocol stacks supporting adaptation

In system design, the capacity to adapt (i.e., adaptability) is determined by the respective architecture and the degree of flexibility it supports. From the viewpoint of software engineering, protocol stack flexibility can be achieved by the following modular design approaches [7]:

1) Adaptable protocols: Adaptable protocols comprise two parts: a generic layer that implements protocol functions common to multiple protocols (e.g., framing) and a custom extension layer that implements additional protocol functions required to realize the function of a specific protocol. From a design viewpoint, this approach resembles significantly that of reconfigurable protocols described subsequently. However, as the smallest unit of design is an entire class of protocol layers (e.g., link layer protocols) the analysis granularity of adaptable protocols is undeniably coarser than that of reconfigurable protocols, where it extends inside as well as across individual protocol layers.

2) Composable protocols: In composable protocols, the overall protocol functionality is analyzed into elementary protocol functions, which are subsequently used as the building blocks for a customized protocol stack. Depending on the particular architecture style and inter-layer communication paradigm, the composition of protocol functions complies with a flat or hierarchical model. However, flexible graph-based approaches are also possible.
3) **Reconfigurable protocols:** Reconfigurable protocols exploit object-oriented design to define the software architecture of the protocol stack and, optionally, support the dynamic replacement of protocol (component) implementations during runtime. The software architecture’s artifacts do not necessarily comply with the conventional stratification paradigm for protocol stacks. Hence, more flexible architecture styles (e.g., [8]) are adopted by some approaches.

### C. Objectives of adaptable protocol stacks

Another classification scheme for adaptable protocol stacks considers their design objective, i.e., which particular goal is pursued by means of an adaptable protocol stack. A survey of the literature reveals the following objectives:

1) **To promote development of protocol stack software:** This objective is concerned with the software engineering and software development issues pertaining to protocol stacks. In the vast majority of cases, the development of protocol stack software is addressed by means of a software framework providing essential protocol stack functions (e.g., inter-layer communication, etc). This category is presented in Section III.

2) **To support the composition of protocols from constituents:** Design approaches that leverage composition in achieving adaptability through protocol modularity are quite common. The unit of composition is typically a subset of the respective protocol’s functionality represented as a module. Modules are composed together with other modules and exchange data either directly and/or through a common facility (e.g., shared memory). However, this does not preclude component-based approaches where coarse-grain units of protocol functionality—or even entire protocols—are packaged into reusable software components. The logical architecture for composite protocols may be also accompanied by a supporting software library providing common protocol functions as prefabricated modules. This extra feature is not an unavoidable consequence of the composite design of protocols, but rather a deliberate effort to ease the burden of developing software components by providing prefabricated solutions to specific—typically common—aspects of overall protocol functionality. This category is presented in Section IV.

3) **To realize competitive performance through customization:** The performance of the communication subsystem and its protocol stacks is a determinant factor of overall network performance. Several research efforts have investigated the effect of software architecture and resource allocation schemes on the performance of protocol stacks. Typically, this problem is addressed by taking into account the applicable resource allocation policies in the design of the protocol stack’s architecture and its supported customization options so as to achieve performance fine-tuning. Protocol stack architectures based on the process abstraction also fall into this category because the respective design choices are fundamentally performance-oriented ones. More specifically, process-based architectures are mapped to thread-per-layer and/or thread-per-message processing models. This category is presented in Section V.

### III. Software Frameworks for Protocol Stacks

With regard to the execution environment chosen for the protocol stack, software frameworks for protocol stacks are classified into the following categories:

1) **Native, where the protocol stacks are executed natively by the operating system in kernel or user space.**

2) **Interpreted, where the protocol stacks are executed in an overlay execution environment (e.g., a virtual machine) that maps their execution to appropriate operating system calls.**

3) **Compilable, where the software structure of protocol stacks is defined in a platform-independent notation that supports software development through formal mappings to native programming languages.**

#### A. Native frameworks

Primarily designed to support the investigation of interprocess communication matters, the x-kernel [9]–[11] is a configurable operating system kernel in which communication protocols define the fundamental unit of composition. In the x-kernel architecture, the exchange of information between protocols is realized exclusively by (variable-size) messages. The x-kernel’s software architecture consists of the following elements:

- Protocol objects with each protocol representing a particular communication protocol (e.g., TCP, UDP, etc).
- Session objects with each session object representing an instance of a communication protocol along with its associated state information and data processing logic. To facilitate control of selected aspects of its operation by a (typically) higher layer entity, session objects provide a set of accessor/mutator operations (e.g., set()/get()) to its internal parameters.
- Message objects with each message object being a generic container for data exchanged between protocol objects and providing a set of management operations to manipulate the data it conveys.

Protocol objects are passive objects (i.e., do not possess their own thread of control) and are statically provisioned as part of the x-kernel’s configuration procedure. Session objects are also passive but are instantiated dynamically during x-kernel operation by protocol objects according to their configuration. Only message objects are active ones (i.e., possess their own exclusive thread of control) according to a thread-per-message processing model. From birth to death, each message in the x-kernel architecture is escorted by an exclusive processing thread that effectively shepherds it through all the session objects (i.e., protocol layers) it traverses.

**Conduit+** [12] is a generic object-oriented framework for protocol software designed primarily to support maximal reusability of protocol software (at the level of software design and the corresponding object code) across different protocols. The Conduit+ framework structure is based on two different classes:

- The conduit class that represents a software component with a pair of distinct connectors.
- The information chunk class for information objects that transit conduit objects.
The conduit class features four types of conduits:

- The Adapter conduit used to interface the Conduit framework architecture to a (hardware or software) resource.
- The Protocol conduit that represents a protocol behavior governed by a specific finite state machine. Typically, Protocol conduits are consumers and producers of information chunk objects carrying protocol (control and payload) data. A Protocol conduit is connected to exactly one conduit object via each of its connectors, thus realizing the layered structure of a protocol stack.
- The Mux conduit which is used to support one-to-many connectivity between conduit objects (i.e., multiplexing and demultiplexing of connections between conduits).
- The ConduitFactory conduit that undertakes the creation or appropriate conduit objects according to the Factory design pattern [13].

Appia [14], [15] is a layered communication framework with extended configuration and programming possibilities that supports the development and execution of modular protocol compositions. In Appia, the interaction of protocol modules is supported by an event-based mechanism with regard to which protocol modules define three sets of events: those accepted by it, those provided by it and those required by it to function properly. The functionality of each protocol layer in the protocol stack is supported by an associated module that realizes the intended service. More specifically, each protocol module is associated to a session object which incorporates state information and provides the respective protocol code as an organized collection of event handlers.

A particular composition of protocols embodies a non-empty set of communication channels, each realized by a protocol stack. The latter is also called a Quality of Service (QoS) as it effectively defines the treatment to which data are subjected as they traverse it. A valid (with regard to QoS) sequence of session instances is termed a channel and implements the intended protocol stack. The Appia system comes equipped with a rich library of protocol implementations that simplify the development of arbitrary protocol stacks through protocol composition. In recent extensions of Appia the dynamic adaptation of a protocol stack involves a) reachability of a so-called quiescent state for communication channels, b) export of relevant state information that must be preserved across reconfiguration from affected modules, c) instantiation of new or additional modules, d) import of state information into the appropriate modules, and, e) resuming of normal operation for the new or additional modules. We note that these elementary steps comply with the well-established design principles for managing dynamic change in software architectures [16]. In Appia, another way of achieving protocol adaptation during runtime is by setting appropriate values to selected operational parameters of a protocol layer. In this case, protocol adaptation is driven and controlled by a specific set of adaptation policies defined according to the Event-Condition-Action (ECA) formalism.

In [17], henceforth termed the GRPSFMT approach, the authors introduce a platform-independent approach for the development of protocol stack software that supports dynamic reconfiguration. As one might expect, the software framework providing the foundation for protocol stack development adopts a modular structure. Composition of protocol stack software is based on a combination of generic and specific protocol functionalities developed using object-oriented techniques and organized in libraries. Besides static (i.e., compile-time) configuration, the software framework supports the dynamic reconfiguration of the protocol stack during runtime. Framework libraries realize the functionality that supports common protocol stack functions (e.g., resource allocation regarding threads, management of timers, etc) as well as the dynamic reconfiguration of the protocol stack.

The framework’s configuration manager entity determines the appropriate configuration of the protocol stacks based on a set of associated XML descriptions and the communication needs of user applications. Message passing and processing between and by the protocol stack’s layers is done according to the thread-per-message model. A dedicated thread is assigned to each message, accompanies it throughout the protocol stack and undertakes the execution of the traversed protocol layer functionalities. A pooling scheme controls and manages the allocation of processing threads to protocol layer messages. Thread management lies under the framework’s control at all times and is employed in the dynamic reconfiguration of the protocol stack to switch protocol layer implementations. The latter is basically achieved by suspending the execution of selected threads, unloading the replaced protocol layers from memory, loading the replacement protocol layers into memory and resuming the execution of selected threads. This approach complies fully to established software architecture designs for managing dynamic software changes [18].

![Fig. 2. The Appia architecture [14], [15].](image-url)
In [19] the authors focus on the reconﬁgurable terminal realm and discuss its reconﬁgurable architecture. The latter is decomposed into ﬁve building blocks: a) the ﬂexible protocol stacks, b) the baseband subsystem, c) the radio front-end subsystem, d) the terminal reconﬁguration management and, e) the middleware layer. Considering the design tradeoffs with regard to component implementation technologies, processing models and software frameworks, [19] proposed a framework to support the runtime reconﬁguration of operational protocol stacks using persistent asynchronous message queues. The framework design is identical to [17], thus we will not detail [19] any further.

In [20] the authors focus on protocol software used for reliable communication over the wireless medium. The objective is to maximize ﬂexibility through a generic (software) framework capable of supporting a wide range of protocols without inducing a signiﬁcant compromise in performance. To this end, similar in purpose protocols (e.g., link layer protocols) are analyzed in detail and their functional commonalities identiﬁed. The results of the analysis support the notion of a generic protocol stack whose functionality can be appropriately extended and/or reﬁned in order to match the function of speciﬁc protocols. The generic protocol stack comprises a) a protocol architecture, b) a protocol framework, c) data structures, d) support for protocol management, and, e) fundamental protocol functions. The latter come in the form of generic parameterizable protocol parts; thus, protocol-speciﬁc behavior is realized by setting appropriate values to selected parameters.

To facilitate reuse across multiple protocols, these generic protocol parts are realized as software modules organized in a software library. The latter effectively provides a toolbox of protocol functions to use in the development of protocol stack software. In addition, the library provides generic constructs to support communication within the realm of an individual protocol layer. A particular protocol’s functionality is realized by combining generic protocol parts with speciﬁc ones that interact using the aforementioned facilities for intra-layer communication. Each composite protocol interacts with its adjacent protocol layers through generic Service Access Point (SAP) primitives. An interfacing module associated to each SAP interprets generic primitives and maps them to protocol-speciﬁc ones supported by the associated protocol’s functional modules. To support dynamic adaptation, each protocol layer incorporates a logical manager entity which supports all administrative tasks conducted during runtime. These include the composition of functional modules, their appropriate parameterization and their organization into a working arrangement that collectively realizes the intended (aggregate) protocol functionality (Figure 4).

[21] presents a software framework for the development of reconﬁgurable protocol stacks based on the composition of elementary functional blocks termed Functional Units. The basic functionality provided by a Functional Unit (FU) is the consumption and/or production of data in variable-sized chunks termed compounds. The data transfer capacity of an entire protocol stack is achieved by connecting appropriate functional units together to match the particular stratiﬁcation of the protocol stack. At the level of an individual FU, interconnection translates to establishing references to inbound and outbound functional units, i.e., that submit and receive compounds to and from this functional unit, respectively. Thus, the resulting network of functional units provides the elementary bidirectional data processing capability that characterizes a protocol stack. Given that a functional unit exhibits all its behaviour reactively in response to compound inputs, the framework is classiﬁed as event-driven from a software engineering viewpoint. As a compound includes functional unit control information in so-called commands, it effectively constitutes an application of the Command pattern [13]. An application of another useful pattern, Chain of Responsibility [13], is recognized in the option of a functional unit to delegate processing of a compound input to another functional unit to which it is directly connected to. Although the construction of a bidirectional data processing capacity that characterizes protocol stacks is supported by the software architecture presented therein, how the latter supports protocol stack reconﬁguration and the particular signaling interactions that realize this remain unclear.
B. Interpreted frameworks

Telecommunication protocols are an integral part of the hosting operating system and are traditionally instrumented using native (i.e., executable) object code that has been compiled for the particular underlying hardware platform. This approach achieves much in terms of performance optimization but requires the complete compilation of protocol software for each target hardware platform. In addition, the resulting object code is tightly bound to the target hardware platform, thus hindering the portability of telecommunication protocol software across different hardware platforms. On the other hand, execution environments based on the virtual machine paradigm (e.g., Java) provide seamless portability of applications across different hardware platforms by delegating and solving all portability-related software engineering matters at the level of the virtual machine architecture. This solve-once-and-for-all approach to the software engineering intricacies of porting software from one hardware platform to another has enabled the widely popular write-once-run-many paradigm of porting software from one hardware platform to another. In addition, the resulting object code is tightly bound to the target hardware platform, for each target hardware platform. In HotLava, each distinct protocol is implemented as a Java class that inherits from a base protocol class specific salient (protocol) features, namely:

- The ability to exploit the dynamic class loading and runtime compilation of the Java virtual machine as technological enablers for the dynamic introduction of protocol software code (e.g., a new version realizing several bug fixes) into the affected communication subsystem during its operation.
- It greatly simplifies the portability of telecommunication protocol software across different hardware platforms, thus achieving economies of scale in telecommunication software development tasks and reducing time-to-market for new telecommunication systems and architectures.
- Enriching the somewhat restricted programming toolset available to communication protocol software programmers with all the powerful features of object-oriented programming (i.e., inheritance, specialization and polymorphism).

These design artifacts underpin the HotLava [22] architecture for protocol stack software. In HotLava, each distinct protocol is implemented as a Java class that inherits from a base protocol class specific salient (protocol) features, namely:

- Data structures common in all protocols regardless of their particular purpose (e.g., buffer management).
- Methods to manage connectivity to other protocol objects by appropriately manipulating the associated protocol graph.

Concurrency of the communication subsystem is supported by a pool of operating system threads according to a thread-per-protocol processing model. A distinct thread is allocated to cater for protocol-specific tasks (e.g., maintenance of protocol state, management of timer timeouts). Furthermore, in the HotLava architecture, each packet is escorted through the entire protocol graph via a non-preemptible thread according to a thread-per-message model, much like the x-kernel case. A particular stack of protocols (e.g., a stack defined in a 3GPP standard) is managed by an instance of a so-called service class that undertakes the instantiation and initialization of all the involved protocol layers as well as their provisioning with state information.

For interfacing to hardware devices and their associated driver software, the HotLava architecture uses a set of special purpose protocols termed adapter protocols that effectively support data exchange across the HotLava virtual machine boundary.

Jgroup/ARM [23] is a middleware platform for distributed Java applications that features a protocol composition framework. In Jgroup/ARM, a so-called group manager encapsulates all the protocol layers associated with a particular application. The underlying composition model adopts a non-hierarchical structure where protocol interaction is realized by means of an event-based mechanism, thereby promoting a loosely coupled interaction that significantly promotes adaptation. The unit of composition is a protocol module. The latter provides a set of services to higher protocol modules (or, in case it resides in the topmost layer of the protocol stack, the application) and requires a set of services from lower protocol layers. Protocol modules services are accessed through an associated service interface. In addition, a so-called listener interface supports the generation of events for other registered protocol modules. The protocol modules employed in a particular protocol stack are dynamically instantiated during runtime and assembled together during the protocol stack’s bootstrap phase.

JChannels [25] leverages the programming features of the Java platform (i.e., dynamic class loading, concurrency, portability) to support the development of configurable modular protocol stacks. JChannels specifies classes for protocol layer and protocol stack through the ProtocolModule and Stack classes. The Message class supports the exchange of information between protocol instances according to a thread-per-message model. To achieve autonomous operation for protocols defined in the context of a protocol stack, an event-based control model is used. The structure of the protocol stack and the identification of the protocols to be included in it are supported by the ProtocolGraph class. A ProtocolGraph instance specifies an ordered list of ProtocolModule instances; hence each Stack instance is associated to a pair of ProtocolGraph instances, one for each direction of data transfer.

In [26], the authors propose that protocol design adopts a clear separation between specification requirements and software (i.e., implementation) logic. This design approach bases the dynamic assembly of the protocol stack on the interpretation of an appropriate description. The latter specifies the desired configuration of the protocol stack. Each protocol layer is defined in terms of two components: a Protocol Description File (PDF) specifying the processing logic, and a Protocol Engine (PE) that actually realizes the protocol’s function at runtime. Based on the PDF contents, the PE dynamically loads the appropriate message parser modules and instantiates the associated finite state machine. To align to common standardization practice, the PDF uses the Augmented Backus-Naur Form (ABNF) as its notation. The accrued flexibility in protocol stack construction comes at the cost of increased delays during the initialization phase, when the object code of individual protocol functions is loaded into the respective execution environment. With regard to software engineering
concerns, [26] applies object-oriented design techniques (i.e., inheritance, encapsulation, etc).

C. Compilable frameworks

By leveraging the pioneering work of the United States Joint Tactical Radio System (JTRS) Joint Program Office (JPO) on software defined radio for defense applications, the Software Defined Radio (SDR) Forum\(^1\) published the Software Radio Architecture (SRA). SRA focuses on the specification of a common framework for building, configuring, connecting and tearing down distributed, embedded radio (i.e., waveform) applications within a (software radio) device [24].

Being based on a variant of the OMG CORBA Components specification [27], SRA defines OMG Interface Definition Language (IDL) interfaces for installing and using distributed waveform applications within a single device. In SRA, a set of XML profiles describe the hardware and software components of an SDR system, their properties and their interconnections (i.e., the software architecture of the SDR device). Thanks to its CORBA foundation, the SRA provides a flexible environment for the seamless integration of heterogeneous hardware and application software developed in different programming languages. The SRA operating environment comprises a) the CORBA middleware and its associated services, b) the SRA Core Framework (CF), and, c) a POSIX-compliant operating system with associated support facilities. Figure 5 depicts the SRA organization and the logical structure of its operating environment. From an OSI model viewpoint, SRA waveforms cover the Physical, Logical Link Control (LLC) and Medium Access Control (MAC) layers and their entailed sublayers, if any.

\(^1\)In general, the scope of software defined radio is restricted to the three lower layers of the protocol stack, i.e., the Physical (PHY), Medium Access Control (MAC) and Network (NET) layers, with particular emphasis on wireless devices.

The OMG Software-Based Communications (SBC) Special Interest Group (SIG) is developing specifications supporting the development, deployment, operation and maintenance of software technology targeted for communication devices defined in software. To this end, the SBC SIG has published a specification for a Platform-Independent Model (PIM) and a Platform-Specific Model (PSM) for software radio systems and components [28]. PIM/PSM is streamlined to OMG Model-Driven Architecture (MDA) modeling conventions and covers a subset of the original SRA model through a so-called platform/waveform approach where the platform provides a standardized yet extensible set of software services that abstracts hardware dependencies and supports development of portable waveform (i.e., software radio) applications as well as other applications types (e.g., management applications). Notably, a portion of the SCA legacy persists in the original SCA model for a software radio device adopted by the SBC PIM/PSM specification. Although the PIM/PSM specification primarily targets the radio subsystem and its associated signal processing functions (e.g., FFT/IFFT, spreading/despreading, modulation/demodulation, etc), it also includes constructs for particular protocol stack aspects. More specifically, PIM/PSM explicitly models common facilities of protocol layers in so-called Building Blocks (BB). These include the Transmission BB providing a mechanism for exchanging data and control information between layers and the Protocol Data Unit (PDU) BB for use in inter-component communications within a software radio and in connectionless communications among different radio sets.

IV. Composable Protocol Stacks

Coyote [29], [30] extends x-kernel and uses micro-protocols that implement individual properties of the target service as separate modules. Micro-protocols are structured using events
and event handlers. This loose coupling approach enhances configurability by minimizing explicit references between modules.

By employing the Coyote runtime framework API the binding of an event handler to a particular event may be changed dynamically during runtime. Event detection and dispatch is provided by the Coyote runtime framework that allocates a distinct thread to process a newly detected event by using the registered event handler. The execution of event handlers typically occurs in parallel (i.e., multiple event handlers may execute concurrently) but sequential execution is also possible. On the other hand, the execution of user-defined event handlers may be either synchronous (i.e., block until all other event handlers in the associated micro-protocol have completed execution) or asynchronous (i.e., proceed regardless of the state of other event handlers). The Coyote runtime framework also provides filtering facilities that allow processing of upstream and/or downstream protocol information by a selected subset of its enclosed micro-protocol objects. The composition of individual micro-protocols into a composite protocol is a static code linking process that takes place when the entire communication subsystem is built from source code.

Cactus [31] addresses issues pertaining to non-functional requirements by employing a fine-grain composition of protocol constructs in a non-hierarchical arrangement that operates under an event-driven execution model. Being similar in design to Coyote, Cactus also supports the construction of configurable network protocols by employing micro-protocol objects as building blocks. In Cactus, each micro-protocol is organized on the basis of possible events and event handlers that influence its particular function and undertake event processing, respectively.

Horus [32], [33] is a portable group communication system based on the Ensemble [34] experience. In the Horus design, a group is an abstract concept that represents a collection of communication peers. The software artifact for the group abstraction supports operations pertaining to membership management (i.e., adding and removing communication peers from the group). With regard to protocol organization, Horus adopts the x-kernel design and supports the graph-based construction of complete protocols from a combination of suitable micro-protocols [32]. In this case, however, micro-protocols are tailored to the specific needs of group communication (e.g., multicast delivery). To customize the basic communication capacity offered by the group abstraction, Horus supports the latter’s extension with additional communication features. Communication features (e.g., flow control, causal order, total order, etc) are packaged into software bundles and can be stratified (i.e., layered) upon each other dynamically during runtime. By stacking a particular set of features, the functional equivalent of protocol layers found in a protocol stack is realized and, thus, applied to group communication.

From a software engineering view, Horus treats each protocol layer as a software module with a pair of standardized interfaces—one to each adjacent (i.e., higher and lower) protocol layer. Object orientation is applied at the level of individual communication endpoints and their groups, as well as at the level of messages. These software constructs serve data modeling and exchange purposes solely (i.e., they do not exhibit proactive behavior of their own). The invocation of protocol behavior is supported by a dedicated process that carries out the respective computational tasks. Hence, from a programming perspective, nested function and/or procedure calls realize the sequential application of protocol-specific behavior across the protocol stack. Notably, a variant of a multi-threaded process model allocated to each protocol layer and combined with an event-based mechanism has also been investigated in Horus. The Ensemble architecture [34] was developed in O’Caml, a variant of the ML programming language providing strong semantics. In Ensemble, protocol services, abstract protocol specifications and software layers are represented as modules specified in a formal notation. This specification is formally checked for correctness and, once verified, used in the automatic generation of software code for the protocol stack according to a synthetic approach.

Da CaPo [35], [36] proposes a logical architecture to support the dynamic configuration of protocol instances in end systems in accordance to application requirements and taking into account the properties of services offered by the network infrastructure. This approach aims to alleviate performance bottlenecks encountered in end systems due to the suboptimal configuration of the protocol stack. To this end, the Da CaPo architecture for end systems comprises three strata: the application (A) layer, the communication (C) layer and the transport (T) layer. The latter represents the transport services of the available network infrastructure as realized by network protocols (e.g., TCP, ATM AAL2/5, etc). The former two comprise end user applications (e.g., instant messaging, video conferencing, etc) and end system protocol layers, respectively. Communication (C) layer services are organized as a composition of protocol functions. A protocol function realizes an elementary aspect of protocol operation encountered in multiple protocols (e.g., encryption/decryption, flow control, error control, bit stream encoding/decoding, etc).

Given that variations may exist between the specifications of protocol functions encountered in multiple protocols, Da CaPo allows for multiple implementations of a particular protocol function as distinct modules. For instance, error control can be realized by single bit parity or cyclic redundancy check codes, depending on the trade-offs between acceptable error rate and induced computational complexity. Da CaPo modules provide for aspect-oriented variants while still capturing the essentials of a protocol function. With regard to composition matters, (C) layer services offered to (A) layer are composed of a set of

---

**Fig. 6.** The Coyote/Cactus architecture [29], [30].
Motivated by the continuing emergence of cross-layer feature interactions in multi-protocol and multi-standard network architectures, [37] puts the layering principle into question and investigates non-layered approaches to the design and implementation of network protocols. The authors argue that, protocol stratification, although undoubtedly beneficial to the overall organization of the communication subsystem through the standardization of inter-layer interfaces, may also undercut its capacity for flexible evolution in the long run. To address this limitation, they propose an architecture, which, instead of protocol layers, employs functional elements termed roles as its constituents. In this so-called Role-Based Architecture (RBA), roles can be organized into arbitrary graph-like configurations, thus providing more options for protocol interaction. Each role processes input payloads and their associated metadata (e.g., protocol information) and may output additional payloads and metadata. By specifying a relatively small set of well-defined roles, the construction of protocol stacks as an assembly of appropriate roles becomes possible. The basic role toolkit can be extended with additional roles developed for specific networking requirements. Basic and extension role objects can be seamlessly combined together into working protocol stacks. This modular approach realizes a significant customization capacity and greatly facilitates the construction of protocol stacks in a ‘mix-and-match’ manner. However, as the ordering enforced by stratification no longer applies in a role-based architecture, the desired sequence of processing by individual protocol functions must be specified through a set of rules.

[38], [39] propose an extensible middleware framework for dynamically reconfigurable protocol stacks. The framework is termed DRAPS (Dynamically Reconfigurable Architecture for Protocol Stacks). Its core component supports the dynamic reconfiguration of peer protocol layers and consists of three components: a) the Reconfiguration Manager (RM) that coordinates reconfiguration signaling to support the switching of protocols between peer layers of the protocol stacks, b) the Stack Factory (SF) responsible for the instantiation of protocol layers from both initial and interim states, and, c) the Execution Environment (EE) that offers the essential runtime services of the protocol stack and supports the execution of protocol instances. In DRAPS, all functions of a particular protocol layer are implemented in a dedicated component. Each protocol component provides an interface to adjacent (i.e., higher and lower) protocol layers through which buffering services for the exchange of user and protocol data are supported. With regard to reconfiguration, each protocol component provides interfaces for a) the control of its computational processes (i.e., basic start/stop operations) and b) the import and export of its internal state. The latter functionality effectively supports the preservation of the protocol’s function across reconfigurations. Transparency of reconfiguration is realized by protocol component wrappers that mediate all communication between interacting protocol components. As DRAPS focuses on the signaling necessary to support reconfiguration and the associated state of affected protocol instances, the internal blueprints of the protocol stack architecture are not detailed (although, based on [38], it most probably uses a thread-per-layer model).

The Distributed Protocol Stacks (DPS) framework [40] abstracts the delay tolerant (i.e., not time critical) functions of a protocol layer as detachable elements that can be deployed into a network node. Through the distribution of protocol functionality the optimization of the protocol stack’s operation is achieved. The design procedure of this framework involves the following steps:

1) Abstraction: this step concerns the selection of functions to abstract. Functions that require minimum or no interactions with the operating system’s kernel structures and the host protocol stack can be regarded as a distinct separate functional unit and, thus, are prime candidates for abstraction.

2) Detachment: this step concerns the detachment of the selected functionality from the host node and its transfer to the target network node. This step may also involve the selection of protocol functionality from a suitable repository.

3) Communication: this step concerns the realization of signaling between the host node and the network node which is realized with the use of a so-called Module Connection Interface (MCI). The MCI includes the components necessary for the internal communication within the network and host nodes as well as the external communication between them.

4) Execution: this step concerns the execution of the detached protocol functionality.

Click [41] presents a flexible software architecture for
router design based on fine-grained functional elements. The basic concept in the Click design is the graph structure of configurations. Specifically, each configuration graph has two basic properties: a) the edges representing possible routes for packet transfer, and, b) the vertices representing functional elements. Each functional element belongs to a particular class (i.e., type) and communicates with other functional elements by means of input and output ports. The Click architecture supports the provisioning of element state and fine-tuning its functionality by employing configuration strings that contain appropriate parameters. In the Click architecture, both push and pull types of connections between elements are supported. Push connections are used when external packets are received and must be processed by a Click element. Pull connections are employed when the receiver of external packets needs to control the actual packet reception time. Furthermore, Click provides facilities for packet storage. Packets are stored in queues realized as functional elements, thus enabling flexibility in the processing of packets through the specification of suitable configurations. Finally, Click includes mechanisms for seamlessly hot-swapping configurations at runtime.

The HIPPARC H project [42] investigated the specification and deployment of a development environment for the automatic generation of protocol modules according to the requirements of distributed applications. Regarding the development of communication protocols, [42] involves a) the specification of protocols in a natural language, b) the validation of protocol specifications by means of formal languages and verification techniques, and, c) the implementation of protocol functions. In HIPPARC H the ESTEREL formal language was employed due to its efficient support for a) protocol descriptions at different levels of abstraction, b) modularity and flexibility in protocol approach, and, c) flexibility in configuration and optimization during protocol software development. ESTEREL is a synchronous language and is employed for the description of the control part of a protocol. Thus, ESTEREL provides a clear separation between the specification and the implementation of a protocol. Notably, the ESTEREL development environment supports validation tasks including the verification and optimization of the protocol description.

Regarding the development of modular protocols, ESTEREL provides parallel and sequential modules that use signals as the communication mechanism, thus offering significant flexibility in relation to the modifications supported by the protocol subsystem. [42] describes the incremental implementation of a protocol based on fundamental protocol functions.

A rather unique viewpoint is adopted in [43] which focuses on discussing alternative approaches for the allocation of network functionalities to protocol layers and their distribution to network elements. Specifically, two approaches are analyzed: a) the horizontal decomposition into network elements, and, b) the vertical decomposition into network elements. In both cases, the objective is to provide a unified framework using a clean-slate top-down design approach for the specification of layered protocol stacks, whilst allowing for functionality distribution and cross-layer resource allocation.

With regard to the horizontal decomposition, [43] presents results of reverse engineering approaches applied against existing protocols in the link, network and transport layers for purposes of optimal protocol stack design. These concepts are applied to the TCP congestion control and random access MAC cases. An example of the reverse engineering approach is the analysis of congestion control as a distributed solution of the basic NUM problem.

V. CUSTOMIZABLE PROTOCOL STACKS

Customizable protocol stacks typically introduce some additional signaling for purposes of protocol reconfiguration procedures. These include dynamic component binding and replacement operations, state management actions and overall coordination signaling. For instance, [8] presents the signaling exchange for end-to-end dynamic binding and replacement of protocol components.

[44] proposed an integrated environment termed ADAPTIVE to develop and experiment with flexible transport system architectures that support adaptable protocols. To this end, ADAPTIVE enables the synthesis of lightweight protocol machines. A Protocol Machine (PM) is an executable object that realizes protocol-dependent processing functions and manages the associated contextual information (e.g., timers, sequence numbers, etc) according to its interaction with a peer PM. To promote performance, ADAPTIVE supports custom schemes for allocating available processes and their associated resources (e.g., threads) to protocol processing tasks. Two process architecture models are supported, namely task-based and message-based. Four possible mappings between process architecture entities and protocol entities are available (Table I). These mappings facilitate the customized and fine-grained allocation of process resources to communication subsystem entities so as to meet performance requirements.

Protocol adaptation is supported at the PM level according to detected changes in the application requirements (e.g., reliable vs. unreliable data delivery), the communication subsystem resources (e.g., CPU load) and, aggregate network state (e.g., congestion). In addition, ADAPTIVE features a repository of software objects realizing elementary functions of common protocol processing tasks. These reusable implementations are used by a formal transformational methodology which, based on application QoS requirements and network characteristics, generates the suitable PM. By utilizing object orientation and dynamic binding programming techniques, the proper PM is dynamically synthesized and instantiated

![Table I: ADAPTIVE PROCESS ALLOCATION SCHEMES [44]](image)

<table>
<thead>
<tr>
<th>Process Architecture</th>
<th>Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task-based</td>
<td>Layer: A distinct process resource is allocated to each protocol layer</td>
</tr>
<tr>
<td>Message-based</td>
<td>Connection: A distinct process resource is allocated to each connection</td>
</tr>
<tr>
<td></td>
<td>Message: A distinct process resource is allocated to each message</td>
</tr>
</tbody>
</table>

NUM is the Monotonic Programming formulation used for reverse engineering TCP variables.
in the ADAPTIVE execution environment. In summary, the ADAPTIVE design targets the following capacities: a) runtime reconfiguration, b) automatic generation of appropriate protocol instances based on formal QoS specifications, and, c) performance efficiency through alternative process architectures and parallelism of execution (i.e., concurrency).

The COMSCRIPT approach [45] takes a different route based on the dynamic interpretation—as opposed to static compilation—of protocol functionality and the application’s ability to define the optimum protocol stack for its needs. From a software architecture view, the COMSCRIPT execution environment is based on the process abstraction. To allow flexibility in implementation, a COMSCRIPT process may realize the function of a single protocol layer or multiple protocol layers (e.g., an entire protocol stack). COMSCRIPT processes are organized hierarchically according to their spawning history and inter-process communication is provided by explicit finite-buffer connectors termed gates. Thus, gates with zero length buffers support synchronous communication while gates with non-zero length buffers provide for asynchronous communication. The programming model of COMSCRIPT processes is that of a Finite State Machine (FSM) whose behavior is driven purely by events. Each COMSCRIPT process is equipped with a set of event handlers and when an event is detected, the appropriate handler is dispatched to handle it. Because a first-class connector object (i.e., the gate) provides for (an essentially message-based) communication between COMSCRIPT processes, the latter are effectively shielded from and unaware of each other. It is this mediation of the gate facility that catalytically enables the transparent reconfiguration of an arrangement of COMSCRIPT processes (i.e., a protocol stack).

In [21], [46] the functionality of an individual protocol layer is composed of a generic part and a specific part that jointly support the particular protocol layer’s purpose. The generic part is common to all protocol layers, regardless of their purpose, while the specific part is unique to each protocol layer and effectively complements the generic part to jointly realize the intended protocol functionality. The analysis process of [46] studies a family of communication protocols to identify their commonalities. The latter are subsequently grouped into generic protocol parts. From a design viewpoint, this procedure is identical to traditional object-oriented analysis that studies the structure of information involved in the problem at hand to identify potential classes and their associated attributes. Though it effectively applies object-oriented analysis techniques in designing its protocol framework, [46] does not clarify whether, at a software architecture level, the combination of generic and specific parts into a valid protocol layer is based on inheritance, composition [13], or both. It is not clear if, in instrumenting a protocol layer, a specific protocol part whose class inherits from the class of a generic protocol part is used as the protocol object or a combination of a generic protocol part with several specific protocol parts is assembled into a composite object that is subsequently used as the protocol object.

[47] recognizes the impact of mobility on the assumptions made about the networking context of end user devices and suggests protocol stack adaptation as a means of dealing with unforeseen changes in the network environment. Notwithstanding the case for adaptation at the application layer (e.g., context adaptation), [47] points out that adaptation strategies confined within a single layer of the protocol stack may not always suffice—or prove efficient in all possible contexts for that matter. It proposes a mechanism based on ICMP messages for propagating information about the network context (i.e., propagation latency, link bandwidth, signal strength, resource costs, etc) to selected protocol stack layers. To support the detection of significant—but not minuscule—changes in the network context, value ranges termed watermarks are associated with each important parameter and an event is triggered upon violation of any of the registered watermarks. The adaptation of transport layer protocols (e.g., TCP, UDP) employs a so-called action table that defines the appropriate action(s) for each watermark. For instance, an adaptation of TCP’s window size can be undertaken by action(s) associated to the violation of the watermark for the end-to-end latency metric.

In [48] the authors address the problem of determining the proper protocol out of a set of given protocols, given a particular communication requirement. To this end, they consider protocol discovery mechanisms based on protocol feedback and identify their essential features. Their approach exploits distinct features of protocol layers to determine which protocol graphs (or protocol paths) can support a given communication task. This is accomplished primarily by trying out different combinations of (peer) protocol instances and interpreting the outcomes to deduce some knowledge about the network configuration. Consequently, this approach addresses protocol interoperability by means of blind identification techniques rather than by a combination of a definitive information model for the identification of protocol features and the support of capability negotiation procedures between protocol instances. However, this approach requires that protocol instances realize explicit failure notifications. In addition, this brute trial-and-error procedure unavoidably introduces some protocol-dependent delays in the discovery process.

[8] presents an approach to the dynamic reconfiguration of the protocol stack that preserves consistency by considering the dependencies that arise between different protocol layers due to the different stratification patterns of multiple system standards (e.g., 3GPP). The structure of protocol software is based on reusable components that are dynamically assembled during runtime in a composite organization that
realizes the intended protocol function. Dynamic adaptation is achieved by switching between interchangeable protocol components and modifying the respective protocol layer structure, if necessary (e.g., to meet additional implementation dependencies introduced by the new component). This process is governed by semantic consistency checks that ensure the compatibility between component interfaces and preserve the overall consistency of the protocol stack. From a software engineering view, the interaction of protocol components is based on queuing facilities that are associated to each protocol component during runtime and subsequently configured to support the communication between adjacent protocol layers. These queuing facilities support the dynamic adaptation of a particular protocol layer by allowing fine-grain control over the interactions that take place between its associated protocol components.

The Accord Programming Framework [49] introduces a component model for the deployment of autonomic grid applications, proposing the separation of composition and computation aspects. This work mainly focuses on the definition of four different concepts, namely a) the application context, b) the structure of an autonomic component, and, with regard to composition, c) rules and d) mechanisms. In defining the application context, Accord builds upon existing frameworks for the definition of functional interfaces, sensors and actuators and proposes the use of SRPC Interface Description Language (SIDL) and Web Services Definition Language (WSDL). The basic elements of the autonomic component in Accord are illustrated in Figure 9. The functionality specified in an autonomic component is provided by three port classes:

1) The functional port that defines the actual functionality supported by the autonomic component in an input-output basis.
2) The control port that includes the sensors, actuators and a set of constraint rules which concern high level policies regarding the invocation of interfaces.
3) The operational port that defines a set of rules used to control either the runtime behavior of autonomic components (i.e., behavior rules) or the different interactions that occur between autonomic components (i.e., interactions rules).

An important aspect of the Accord programming framework lies in the definition of dynamic component composition. The associated operations in support of adaptation are realized on a functional port basis and are orchestrated by a composition agent. The latter defines the appropriate composition and/or interaction rules that are then injected in the respective rule agents. The rule agents apply these rules and the respective configuration, thus controlling the composition of autonomic components. We note that the component replacement process is also handled by the composition agent. The latter injects the interaction rules to the new component and informs the other components in the specified configuration about the update in the interaction rules. In addition, the composition agent handles the transfer of state information between the replaced and the replacing components. Unfortunately, [49] does not detail the interactions and information elements exchanged between Accord components and their management entities during composition and replacement procedures.

Another category of frameworks considers an entire protocol layer as a component. This design approach is applied in DiPS/CuPS [50], [51], which allow the dynamic adaptation of protocol stacks in order to accommodate various application demands or network optimization objectives. The Distinet Protocol Stack Framework (DiPS) is a component-based framework for the deployment of open multi-threaded protocol stacks, allowing at the same time for flexibility in defining the appropriate configuration of the protocol stack. Customizable Protocol Stacks (CuPS) is considered a meta-level extension of DiPS with support for unanticipated dynamic customizations. We note that DiPS/CuPS do not describe in detail the adaptation realization. Merely a single CuPS case study involving the analysis and development of the CRP (Customizable Reliability Protocol) protocol is provided by the DiPS/CuPS literature. In particular, the CRP protocol has a predefined customization area to allow the introduction of additional so-called MultipleSender components whenever the underlying communication links exhibit considerable packet loss. Consequently, the supported adaptation capacity is not a dynamically configurable (e.g., based on the instant application or service requirements) but a static one that can only be applied in predefined unanticipated cases.

Components are also the basic unit of protocol composition in the THINK and Fractal frameworks. FRACTAL [52] introduces a generic recursive component model which allows sharing of sub-components between components. Fractal provides a generalized recursive component model for the development, deployment and management of complex operating systems, including their communication subsystem and its enclosed protocol stacks. FRACTAL provides dynamic adaptation capabilities through a generic component model that supports composite components to enable various abstraction layers. FRACTAL distinguishes two component types: composite components, which are composed of other sub-components, and primitive components. In both types, the internal component structure is identical and includes a controller and the content. Each component also provides two interface types: a client interface for invoking its operations and a server interface for management purposes.

Protocol component binding/unbinding procedures are based on the BindingController interface. The FRACTAL component model also takes into account the requirements for safe component replacement by providing a component interface for lifecycle management purposes. THINK is another component-based framework for building flexible operating
system kernels and also applicable to protocol stack adaptation [53]. THINK enables the assembly of operating system kernels during operation or runtime based on appropriate configuration graphs. This framework provided an integrated architecture for the aforementioned functionality. In THINK all entities are implemented as components, based on KORTEX, a library of kernel components. A component’s implementation includes both data and behavior. Several types of components are implemented according to the type and number of interfaces; for example, a component implementing a unique interface in a specified domain is called a singleton. The realization of component bindings is supported by a Binding Factory mechanism [13]; thus, the generation of implicit (indirect) bindings is possible (e.g., using Java RMI). In KORTEX, several types of bindings are supported including ‘local’, ‘syscall’, ‘upcall’ and ‘signal’ bindings. In addition, synchronous local PRC and remote RPC bindings are available. Another valuable feature of THINK is that configuration is built using component-based graphs. A case study of THINK investigated whether performance bottlenecks that exist in Solaris and Linux kernels manifest in THINK as well. The evaluation indicated that the use of the KORTEX kernel increased throughput more than 30% compared to the other kernel options. We note that THINK is actually a C language implementation of the FRACLAL framework [53].

The Performance-Oriented Reference Model (POEM) [54], introduces a somewhat different vision on the organization of communication software and the design of a protocol stack by incorporating autonomic aspects. The benefits of holistic and systematic cross-layer optimization form the core of this work, building upon traditional concepts such as the protocol heaps. The POEM model incorporates a Common OptimisatiOn Layer (COOL) which provides the different autonomous and self-organizing capacities and realizes the interactions with the different protocol layers through so-called Common Optimization Interfaces (COIN). Adopting the traditional plane-based approach for networking design, the POEM model introduces two different planes: a) the user plane which realizes the data flow interactions, and, b) the control plane which realizes the control functionality featuring the self-organization interactions with the use of the COOL layer and its associated COIN interfaces. Inspired by the layered natural ecosystem structure and the interactions and interdependencies of the entities in each layer, the POEM model proposes the integration of Critical Control Points (CCPs) and sensor entities for the gathering of context data related to the protocol stack. Such data are subsequently analyzed and used to predict as well as drive the behavior of the protocol stack based on specific policy management rules. Furthermore, actuators are specified and used to implement specific actions. This concept has been extended to a so-called Composable Functional System (CFS) that can be composed and reconfigured out of a set of self-optimizing functional elements. The application of POEM to the protocol stack can lead to the optimization of parameters, such as delay, jitter and loss rate.

A terminal architecture capable of adapting to the mobility protocol offered by the wireless access network is presented in [55]. The architecture is based on the decomposition of...
the mobility protocol’s services into a number of orthogonal components. This work proposes a modular architecture for the mobility protocol. Adaptation concerns the dynamic alteration of the mobility protocol’s flavor by allowing for the switching between micro and macro mobility protocols. In engineering the aforementioned flexibility, the first step is the analysis and decomposition of protocol functionality into distinct parts. The objective is to identify common parts among the mobility protocols in order to minimize the parts that should be altered by the reconfiguration procedure. This work also discusses design issues that should be addressed to support reconfiguration of the mobility protocol, namely:

- The enhancement of the system architecture with support for multiple mobility protocols. For instance, in the case of Mobile IPv6, the introduction of an additional binding acknowledgement packet was proposed.
- The sequence of protocol invocation. This was investigated by examining the relations and interoperation of operating protocols.
- The management of the routing table. This is related to the manipulation or routing rules and their dynamic adaptation to the latest network settings. To this end, a suitable mechanism was proposed in [55].

The proposed concepts were evaluated by implementing the respective architecture and evaluating its performance under the simultaneous operation of micro and macro mobility protocols.

DiPS+ [56] proposes a software architecture to support flexibility in the protocol stack design as well as management of protocol functionality and concurrency issues (e.g., processing of packets which cannot be realized within a specific time bound). The basic concept of this architecture is the introduction of two planes: a) the data plane that handles the protocol stack functionality and its implementations as a collection of various protocol layers and protocol components, and, b) the management plane which is responsible for the management and control of the behavior of the data plane. More specifically, the data plane includes the following architectural styles:

- Pipe and filter style: This is realized with the use of incoming and outgoing packet flows. A connector component supports the exchange of messages between different components whereas a dispatcher component acts as a message demultiplexer at each protocol layer.
- Blackboard style: this architectural style dictates the use of messages between communicating entities. When a task needs to be fulfilled, a message is forwarded to the input communication queue of the component that should realize this task. A data structure (called blackboard) is also used to hold state information about the components, thus allowing anonymous interactions between them to occur and facilitating component reuse and adaptation in different protocol stacks formations.
- Layered style: this concerns the adoption of the traditional layered design for protocol stacks.

The management plane deals primarily with concurrency, load management and the interception of packets. Concurrency management is realized with the incorporation of concurrency components, used either for adaptation or for scheduling the processing of packets. Load management facilitates the global optimization of system performance as well as the control of the requests arrival rate by using priority in the incoming packets. The validation of these concepts was realized by implementing the Remote Authentication Dial-in User Service (RADIUS) protocol and introducing concurrency components. This case study validated the component-based design concepts that allowed the dynamic differentiation of user and request types. User differentiation in RADIUS is realized by inserting a new component in the protocol, the concurrency component, which provides user classification in three classes: gold, silver and bronze. In a similar manner, requested type differentiation is realized with the introduction of two concurrency components before the authentication and the accounting layer. Thus, the adaptation of the RADIUS function is realized.

DROPS (Dynamically Reconfigurable Protocol Stacks Project) developed a framework that facilitates the implementation of modular protocols [57]. In addition, the framework supports the dynamic adaptation of protocol functionality during runtime. In DROPS, every protocol is formed out of microprotocols. Protocols and microprotocols are formally defined by means of a BNF notation. The latter is also used for the configuration of the protocol stack. In the DROPS architecture, a so-called shepherd function handles the processing of protocol payload data and the sequential execution of microprotocols. However, it is the responsibility of each microprotocol to invoke the subsequent (with regard to processing order) microprotocol. The modification and configuration of protocol stacks in DROPS is realized by means of include, exclude and exchange operations. For example, the include and exclude operations allow the incorporation and removal, respectively, of a microprotocol in a protocol stack.

Configuration consistency is an important technical aspect investigated in DROPS. Specifically, the DROPS design and architecture requires that identical configurations should be maintained at all communication endpoints. This requires that the same microprotocol is used in peer protocol layers and in identical data processing order. The solution proposed by the DROPS system is based on the following principles: a) the unique identification of each microprotocol and, b) the exchange of the configuration used for data encoding. Moreover, to ensure consistent configurations, two mechanisms are proposed: a) a dedicated connection to use in conveying the necessary configuration information, and, b) to communicate configuration information within the outgoing data segments. Regarding compatibility, existing applications may utilize both legacy protocols and adaptable protocols without requiring any modifications. To this end, DROPS supports protocol masquerading and enables routing of incoming data segments to the appropriate protocol stack. The proposed mechanisms were prototypically implemented and a custom protocol was used in validating the concepts as well as the architecture. The performance evaluation concluded that the dynamic adaptation of protocol functionality can seriously affect the performance of running applications.

Scout [58] proposed an Operating System (OS) that exploits a path-based design for network devices. A path is viewed as
a logical channel used for data flows and the provision of access to the required context through multi-layered systems. The Scout design builds upon two basic design principles: a) the development of explicit path abstractions, and, b) the use of invariants to describe path properties and facilitate access to global context. The creation of a path is realized through function invocation and the definition of the associated invariants. Instead of using the typical bidirectional model, each path has two distinct queues (one per direction) to decouple the arrival of messages from the execution of path functions.

The path abstraction is also the basic concept of the Scout OS that uses routers as the basic architecture unit. Each router provides a set of services that may be offered to other routers. Routers are initialized in a manner that guarantees the correct execution of a service. Path creation takes place incrementally as a sequence of stages that connect a pair of services. Data routing across paths is based on multiplexing and demultiplexing facilities. A provisioned set of paths is initialized during system bootstrap to handle network packets. The configuration of Scout routers is done during build time; consequently, the associated configuration graph cannot be extended during runtime. An alternative proposed in [58] is to support extensibility by including an interpreter into the router graph. Experiments proved that the performance of Scout is comparable to that of the Linux operating system for applications involving the reception, decoding and display of MPEG video.

Table II summarizes the presentation of Sections III, IV and V by listing the distinctive features of adaptable protocol stack approaches. As in some cases, the bibliography did not detail all technical aspects, we had to resort to deduction based on the overall presentation provided. When it was impossible to do so in a sufficiently confident manner, the tag N/A (i.e., not available) was used for the respective attribute.

VI. DISCUSSION AND LESSONS LEARNED

When we consider the different design objectives of adaptable protocol stack frameworks, we identify six key performance indicators, which, collectively, can be used to assess the overall suitability of each framework for a particular application. These factors include the system design externalities, the ease with which individual protocols and/or protocol components can be composed into a complete protocol stack and protocol layer, respectively, the flexibility in the configuration of arbitrary protocol stacks, the architecture’s openness and its support for unanticipated extensions of functionality, the complexity of protocol stack formation, and, last but not least, the performance attained during protocol stack operation.

A. System design

The merits of component-based approaches are indubitably determined by the amount of design insight achieved during the associated analysis phase. Approaches based on the strict stratification of protocol layers into a protocol stack inevitably bear the limitations associated to this rigid design (e.g., inability to support adaptations by means of cross-layer interactions). On the other hand, approaches that adopt a flexible graph-based definition for the sequence of layer processing [8] offer significant flexibility in achieving adaptation through suitable modifications of the protocol graph (e.g., by modifying the processing of existing layers and/or introducing additional ones). However, this comes at the cost of an increased difficulty in correctly identifying the particular function of protocol layers and properly delineating their boundaries in the protocol graph [59]. Because this flexibility may prove detrimental to the development of the respective product market in the long run, prudence and caution must be exercised in protocol stack design according to graph-based approaches. Modeling concepts and design techniques from the field of software architecture may contribute in efficiently confronting these challenges.

Stack-like approaches based on the composition of protocol functions into a working protocol layer are by design unable to support adaptations where the desired protocol behavior cannot be decomposed into elements that fit a stack pattern of arrangement. This limitation holds regardless of whether the desired behavior is a variant of existing protocol behavior or an entirely new one.

B. Complexity of protocol stack formation

The complexity of protocol stack formation relates to the definitive specification for the stratification of its constituent protocol layers and their internal organization, if any (e.g., in the case of composable protocols). Furthermore, the definition of cross-layer interactions between non-adjacent protocol layers may complicate protocol stack formation. In this respect, the support of cross-layer interactions may introduce additional complexity. For instance, the realization of cross-layer interactions based on the Conduit+ framework is expected to be a more complex design exercise than, say, by employing the Coyote and Cactus frameworks. The difference in complexity stems from the different styles of interaction between non-adjacent protocol layers—explicitly specified connections between Conduit+ objects versus implicit interactions by using shared memory facilities. In general, software development frameworks for protocol stacks are associated with a higher complexity in protocol stack formation than composable protocol stack approaches.

In the Coyote and Cactus frameworks, protocol behavior manifests through its collection of event handlers. Hence, the management of event handlers provides a versatile mechanism for the flexible manipulation of the entire protocol stack during runtime. Albeit this approach presupposes an expert knowledge of protocol behavior, its engineering instruments (i.e., events and event handlers) are simple yet powerful and efficient. The DPS approach supports the migration of protocol functions between execution environments residing at different network elements—a task that is complex in design as much as in realization.

Nonetheless, the assessment of complexity entailed in a dynamically adaptable protocol stack is far from straightforward. All the frameworks presented herein are designed with performance requirements in mind and, thus, support concurrency according to a thread-per-layer and/or a thread-per-message model. Consequently, differences in complexity...
### TABLE II
**Comparison of dynamically adaptable protocol stacks**

<table>
<thead>
<tr>
<th>Approach</th>
<th>Protocol graph</th>
<th>Specification</th>
<th>Processing model</th>
<th>Programming model</th>
<th>Software unit</th>
<th>Composition unit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Adaptable protocol stacks</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conduits+</td>
<td>per stack</td>
<td>static</td>
<td>thread-per-stack</td>
<td>event-based</td>
<td>module</td>
<td>protocol</td>
</tr>
<tr>
<td>JChannels</td>
<td>per stack</td>
<td>static</td>
<td>thread-per-message</td>
<td>message-based</td>
<td>object</td>
<td>module</td>
</tr>
<tr>
<td>CuPS</td>
<td>per stack</td>
<td>static</td>
<td>thread-per-stack</td>
<td>event-based</td>
<td>module</td>
<td>module</td>
</tr>
<tr>
<td>On... [47]</td>
<td>per stack</td>
<td>static</td>
<td>thread-per-stack</td>
<td>event-based</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>SCOUT</td>
<td>per path</td>
<td>static</td>
<td>thread-per-protocol</td>
<td>event-based</td>
<td>path element</td>
<td></td>
</tr>
<tr>
<td>HIPPARCH</td>
<td>per stack</td>
<td>static</td>
<td>thread-per-protocol</td>
<td>event-based</td>
<td>module</td>
<td>module</td>
</tr>
<tr>
<td>POEM</td>
<td>per stack</td>
<td>static</td>
<td>thread-per-protocol</td>
<td>event-based</td>
<td>protocol</td>
<td>protocol</td>
</tr>
<tr>
<td><strong>Composable protocol stacks</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Appia</td>
<td>per stack</td>
<td>static</td>
<td>thread-per-stack</td>
<td>event-based</td>
<td>module</td>
<td>protocol</td>
</tr>
<tr>
<td>x-kernel</td>
<td>per class</td>
<td>static</td>
<td>thread-per-message</td>
<td>method invocation</td>
<td>module</td>
<td>micro-protocol</td>
</tr>
<tr>
<td>Ensemble</td>
<td>per group</td>
<td>N/A</td>
<td>thread-per-protocol</td>
<td>event-based</td>
<td>feature</td>
<td>feature</td>
</tr>
<tr>
<td>Coyote/Cactus</td>
<td>per stack</td>
<td>static</td>
<td>thread-per-protocol</td>
<td>event-based</td>
<td>module</td>
<td>module</td>
</tr>
<tr>
<td>Horus</td>
<td>per group</td>
<td>N/A</td>
<td>thread-per-protocol</td>
<td>event-based</td>
<td>feature</td>
<td>feature</td>
</tr>
<tr>
<td>ADAPTIVE</td>
<td>per object</td>
<td>static</td>
<td>thread-per-protocol</td>
<td>event-based</td>
<td>object</td>
<td>protocol function</td>
</tr>
<tr>
<td>DPS</td>
<td>per stack</td>
<td>static</td>
<td>thread-per-protocol</td>
<td>event-based</td>
<td>protocol</td>
<td>protocol function</td>
</tr>
<tr>
<td>RBA</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>role</td>
<td>role</td>
<td></td>
</tr>
<tr>
<td>HIPPARCH</td>
<td>all stacks</td>
<td>dynamic</td>
<td>thread-per-protocol</td>
<td>push/pull</td>
<td>protocol</td>
<td>protocol</td>
</tr>
<tr>
<td>Da CaPo</td>
<td>per service</td>
<td>static</td>
<td>thread-per-protocol</td>
<td>N/A</td>
<td>module</td>
<td>module</td>
</tr>
<tr>
<td>DiPS</td>
<td>per stack</td>
<td>static</td>
<td>N/A</td>
<td>N/A</td>
<td>component</td>
<td>component</td>
</tr>
<tr>
<td>DiPS+</td>
<td>per stack</td>
<td>static</td>
<td>N/A</td>
<td>N/A</td>
<td>component</td>
<td>component</td>
</tr>
<tr>
<td>DROS</td>
<td>per stack</td>
<td>dynamic</td>
<td>thread-per-protocol</td>
<td>message-based</td>
<td>module</td>
<td>micro-protocol</td>
</tr>
<tr>
<td><strong>Reconfigurable protocol stacks</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HotLava</td>
<td>per object</td>
<td>dynamic</td>
<td>thread-per-protocol</td>
<td>method invocation</td>
<td>object</td>
<td>protocol</td>
</tr>
<tr>
<td>COMSCRIPT</td>
<td>per stack</td>
<td>dynamic</td>
<td>thread-per-protocol</td>
<td>event-based</td>
<td>process</td>
<td>Finite State Machine (FSM)</td>
</tr>
<tr>
<td>SCA</td>
<td>per stack</td>
<td>static</td>
<td>thread-per-component</td>
<td>method-based</td>
<td>component</td>
<td>component</td>
</tr>
<tr>
<td>GRPSFMT</td>
<td>per stack</td>
<td>static</td>
<td>thread-per-message</td>
<td>N/A</td>
<td>object</td>
<td>protocol</td>
</tr>
<tr>
<td>DRAPS</td>
<td>N/A</td>
<td>N/A</td>
<td>thread-per-protocol</td>
<td>N/A</td>
<td>protocol</td>
<td>protocol</td>
</tr>
<tr>
<td>Alonistioti [8]</td>
<td>per stack</td>
<td>dynamic</td>
<td>thread-per-protocol</td>
<td>message-based</td>
<td>component</td>
<td>component</td>
</tr>
<tr>
<td>THINK/FRACTAL</td>
<td>per object</td>
<td>dynamic</td>
<td>thread-per-component</td>
<td>message-based</td>
<td>object</td>
<td>component</td>
</tr>
<tr>
<td>ACCORD</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>component</td>
<td>component</td>
</tr>
<tr>
<td>Reconfigurable IP Mobility [55]</td>
<td>per-protocol</td>
<td>static</td>
<td>thread-per-protocol</td>
<td>event-based</td>
<td>component</td>
<td>component</td>
</tr>
</tbody>
</table>

have to do mostly with the impact in matters of development, configuration and management. Clearly, software framework approaches address primarily protocol software development issues. Not surprisingly, composable protocol stacks fare well in all these matters. Reconfigurable protocol stacks introduce additional complexity in the engineering of configuration and management procedures to realize the capacity for dynamic adaptation during runtime.
C. Composition model

With regard to composition, Coyote and Cactus support a general model for composing software modules into working arrangements, while Horus and Ensemble support hierarchical or stack-like arrangements only. In Coyote and Cactus, protocol layers are organized hierarchically, just like in the x-kernel case, while within each (composite) protocol layer, micro-protocols interact as peers to each other in an arbitrary organization. In the x-kernel framework, the subjects of composition are individual protocols and the outcome of composition is a complete functional protocol stack. In the Coyote and Cactus systems, the subjects of composition are, in addition, micro-protocol modules that implement individual properties of the protocol service and the outcome of composition is a complete functional protocol. Furthermore, in Coyote, the mechanism for event detection and dispatch supports the communication between micro-protocol objects within the confines of a composite protocol (i.e., a single protocol layer). On the other hand, Ensemble employs events under a FIFO arrangement as the communication mechanism between multiple protocol layers realized by micro-protocol objects. In contrast to Horus and Ensemble, Coyote and Cactus support shared data objects for micro-protocols.

Coyote also acknowledges the dramatic increase in flexibility offered by the dynamic composition of micro-protocols during runtime. It also acknowledges the ungratifying discrepancy between the theoretically possible and the practically limited reusability of micro-protocols across multiple protocols. This limitation is due to differences in the overall algorithmic framework imprinted in each particular protocol’s function that—inevitably—affects the definition of micro-protocol functionality, thus limiting their reuse.

Appia basically extends the Ensemble design by supporting the composition of multiple protocol modules into a single protocol layer. However, in Appia, the channel design concept enforces a strictly hierarchical composition that limits the dispatching of events to adjacent protocol layers of the protocol stack, thus restricting the possible interaction patterns between protocol components and protocol layers. Contrarily, the Coyote and Cactus event dispatch mechanism supports the interaction between arbitrary micro-protocols, not just those composed into the same protocol instance—or adjacent layers of the protocol stack, for that matter. In other words, cross-layer design applications are not precluded by Coyote and Cactus. In addition, Coyote and Cactus support the interaction of micro-protocols through the data sharing facilities provided by their runtime. Hence, the protocol interaction capacity offered by Coyote and Cactus is a genuine superset of that of Appia. Consequently, Coyote and Cactus support cross-layer interactions while Appia does not.

In Accord, the composition model is based on the introduction of composition agent functionality and interaction rules in the components. The composition agent defines the component configuration and composition pattern and subsequently applies it by specifying the associated component interaction rules which are injected in the composed components. The same procedure is applicable to both the component binding and replacement procedures. FRACTAL and THINK support interface-based component composition. Component binding and unbinding is realized by invoking the BindingController interface. It should be noted that these frameworks support seamless component replacement (i.e., without loss of data) by incorporating mechanisms for the safe (with regard to state consistency) replacement of components.

The Reconfigurable IP Mobility component framework [55] puts emphasis on the decomposition of protocol functionality into orthogonal components in order to derive common functionality that could be reused across protocol variants. However, the emphasis is paid on the adaptation of the terminal architecture for switching from micro to macro mobility and the component composition/replacement is not investigated thoroughly. Merely a proposition is given that component replacement can—potentially—be realized with the use of C pointers or method overriding in object-oriented languages.

D. Flexibility of configuration

In Coyote and Cactus, the use of a first-class abstraction (i.e., the event class) for micro-protocol interaction provides a light decoupling mechanism that significantly promotes flexibility. Thus, the collection of micro-protocols forming a composite protocol is dynamically amendable during runtime without affecting the operation of other micro-protocols. As a general principle, the objectification of concepts involved in inter-protocol communication provides for the decoupling of their associated interactions, thus enabling the dynamic change of the protocol stack during runtime [16]. This fundamental design pattern underpins middleware-based approaches as well (i.e., SCA, SBC PIM/PSM).

From a design view, x-kernel exhibits several deficiencies, including a lack of support for complex interactions among protocol objects, limited facilities for data sharing, and an orientation towards hierarchical protocol composition at the expense of more flexible combinations.

With regard to component-based approaches, DiPS does not address the functional requirements of dynamic configuration adaptation (i.e., during the runtime operation of the protocol stack). This essential feature is however supported by CuPS which, is considered as a meta-level extension of DiPS and allows for dynamic unanticipated customizations in terms of protocol component addition, removal and replacement.

In Fractal, the recursive model of component composition dictates that, during reconfiguration, the operation of all affected components as well as their communication peers are suspended. Hence, in Fractal, the replacement of a single protocol component may have an extensive effect on the normal operation of the protocol stack by requiring the interruption of a large number of protocol layers’ operation. Most importantly, this introduces a considerable amount of additional signaling throughout the protocol stack and may significantly prolong the time it takes for reconfiguration to complete.

Regarding platform support, middleware-based approaches (i.e., SCA and SBC PIM/PSM) offer the greatest flexibility. Thanks to its CORBA foundation, SCA-based protocol stacks can be developed in any suitable programming language and execute on dissimilar (in terms of hardware and/or software) systems. Nonetheless, the CORBA architecture underlies the
structure of all SCA protocol stacks. This limitation is raised by the SBC PIM/PSM approach that complies to the modern OMG DMA design principles. The latter promotes a generic design which, in addition to being independent of operating system and programming language, supports the use of any suitable component technology (e.g., Enterprise JavaBeans, CORBA, .NET) for the software architecture.

DROPS supports the introduction of flexibility in protocols. It should be noted that after the protocol adaptation procedure, DROPS supports two options for the selection of a suitable protocol stack by applications: either the existing protocol stack can be used or an adaptable one should be employed. In the latter case, DROPS provides protocol masquerading and packet routing techniques so that the information is processed by the appropriate protocol stack.

[55] provides supporting mechanisms for the reconfiguration of the mobility protocol as architecture enhancements, i.e., the introduction of a Mobile IPv6 binding acknowledgment packet. This framework introduces great flexibility in the dynamic adaptation of the mobility management protocol by enabling the alteration of micro and macro mobility protocols during runtime.

E. Openness of architecture

Openness concerns unanticipated extensions of the system in question and the relative ease with which they can be pursued, defined and realized. For instance, both Ensemble and Appia employ an event-based mechanism for the interaction between protocol modules; however in Ensemble, the set of possible events is a statically defined one, while in Appia it is possible to define additional event types by extending the base class for events. The merits of event-based interaction patterns are also realized in the Coyote and Cactus frameworks.

Design approaches based on object orientation (e.g., SRA) are inherently extensible, at least with respect to the definition of unanticipated information classes as extensions of existing ones (i.e., with regard to data modeling matters). However, the extent to which inheritance and polymorphism support the definition of behavioral variations depends on the respective software architecture and, more specifically, the use of appropriate design patterns [13]. This holds true for all frameworks with object-oriented design foundations (e.g., HotLava, JChannels, etc). For instance, the experience gained from a prototype development of the IPSec protocol using the Conduit+ framework lead to the following findings:

- The usefulness of the ConduitFactory class is significantly increased when its design is revised according to the Abstract Factory design pattern so that arbitrary types of conduits can be created.
- The level of software reusability facilitated by the Conduit framework is maximally realized when overall protocol functionality has been thoroughly analyzed and decomposed into elementary functional parts.
- Being a black-box framework for network protocol software, the Conduits+ framework provides common protocol structure and behavior in reusable software components which, can be combined into different protocol implementations without any source code modifications.

FRACTAL provides a generic and extensible framework for component definition, composition and replacement which can be employed by different types of dynamically adaptable systems and applications. The development and availability of several FRACTAL implementations in different programming languages (e.g., THINK is an implementation of FRACTAL in the C programming language) stands as testimony to the wide applicability of FRACTAL. In addition, the initial FRACTAL version has been gradually extended to in terms of the provided functionality (e.g., by defining functional extensions to support component-based control membranes).

Openness of architecture is an important feature of the DIPS system, which was further extended to support dynamic unanticipated customizations of the protocol stack. The extended system was called CuPS. DiPS+ is another extension of DiPS which handles both flexibility and concurrency requirements in protocol adaptation.

F. Performance

The typical processing model for protocol stacks is that of a single dedicated process which reacts in response to stimuli (e.g., method and/or procedure invocation, submission of an event, etc) from its adjacent protocol layers. Because method and/or procedure calls entail a context switch in the processor execution stack, they may adversely impact performance. In addition, if multi-threaded operation (i.e., concurrency) is supported within a protocol layer, cautious design measures are imperative to provide exclusive access to critical code sections and to avoid deadlocks and livelocks. Admittedly, these performance deficits plague the Horus design [34]. To alleviate the burden of thread synchronization from protocol software developers, Appia adopts a thread-per-stack model where a single thread undertakes the processing of all the events that occur in the context of a single protocol stack. Unfortunately, this design does not lend itself well to functional parallelism and performance. In Coyote and Cactus, the processing of events occurring in a composite protocol is not a critical task; hence multiple events may be processed concurrently within a composite protocol. However, while Coyote and Cactus allow for the concurrent processing of events, their runtime services do not enforce exclusive access to critical resources (i.e., synchronization is the responsibility of the software developer). Notably, both these systems support the association of multiple handlers to a particular event which, when raised, results in their sequential execution.

In middleware architectures (e.g., SRA) the involvement of several software layers (e.g., object request broker, object adapter, etc) in protocol data processing is a determinant factor in achieving satisfactory performance. With regard to performance, the SRA operating environment supports custom-off-the-shelf (COTS) hardware and common bus architectures (e.g., VME, cPCI, etc). However, the choice of hardware artifacts for an SRA implementation is ultimately determined by the performance requirements (e.g., timing constraints) of waveform applications (e.g., GSM, GPRS, etc). An experimental investigation of the SRA performance on a generic purpose processor platform revealed that at most
20% of the overall processing delay was due to the mediation of the CORBA middleware. By careful engineering and performance optimizations (e.g., client/servant collocation), minimal SRA delays on the order of a few microseconds can be achieved on modern generic purpose processor platforms [6]. The OMG work also supports performance optimizations through CORBA standards for specific application domains. In response to the increasing demand for CORBA applications on embedded platforms, OMG has developed the CORBA/e specification [60]. The latter preserves the established CORBA interoperability and, by including essential features of the Real-Time CORBA specification, provides real-time predictable performance.

In DROPS, the introduction of flexibility in dynamic protocol adaptation and configuration consistency seriously affects the performance of the running applications. The result of performance evaluation studies with a custom protocol, the Random Adaptable Protocol (RAP) providing for adaptable acknowledgement schemes, proved that, compared to static control configuration, the message delivery delay experienced by applications is much greater when dynamic configuration schemes are used.

The key concepts of the Accord programming framework have also been prototyped for purposes of performance evaluation. Such concepts mainly concern the dynamic injection and execution of the appropriate rules. The outcome of these performance studies confirmed that the introduced functionality has a negative impact execution time and system performance; however [49] concludes that the additional delay is tolerable.

Although [50], [51] provide no performance evaluation results, its extension, CuPS, does. The performance evaluation of the CuPS framework was realized by evaluating the operation of the CRP protocol. However, the CuPS studies mainly compare the performance of the replaced (i.e., old) and the replacement (i.e., new) protocol component in terms of delay and jitter improvements or network load and do not investigate how the introduction of flexibility affects system operation (the same comment applies to DiPS+). In the latter case, performance evaluate studies using the RADIUS protocol favor DiPS+ due to the minimal additional overhead (less than 5%); a rather small price to pay in exchange for the flexibility it offers in differentiating user and request types.

A case study of the concept proposed in [40] trialed the detachment of the TCP ACK functionality from the mobile host and its relocation at the radio base station using an ARQ proxy. Simulation of this case study in a IEEE 802.11 radio setting demonstrated an improved system performance in terms of the delay experienced on the TCP receiver feedback channel and the tolerance to wireless link errors. However, this approach has serious limitations in cases of high user mobility and in systems with dynamically organized network nodes.

Regarding the performance of the Click architecture, [41] includes performance evaluation results for IP routing and several extended configurations. The proposed modular architecture imposes two types of overhead: a) overhead related to the passing of packets between elements which is due to the involvement of virtual function calls, and, b) overhead due to generic element code which is not necessary. The first type of overhead can be avoided by minimizing the use of virtual function calls while the second type of overhead is relatively small and does not significantly affect the overall system performance.

In [55], the performance assessment studies deal with parallel protocol operation and processing (i.e., MIPv6 together with BCMIPv6 and for MIPv6 only). These studies proved that system operation is not affected by the concurrent operation of different mobility protocols. However, we should note that network performance is essentially enhanced compared to the traditional case where a single mobility protocol operates.

VII. CONCLUSIONS

The introduction of dynamic adaptation capacities in protocol stack software has been intensively investigated by several research efforts. The latter differ substantially with regard to the extent of feasible adaptations, the software architecture style for the protocol stack and the protocols it includes, the granularity of composition, the applicable processing model, the programming model that supports software development tasks and the amount of parallelism in protocol processing. These design parameters determine the adaptation capacity and influence the performance of the protocol stack. Hence, judicious choices in design are of paramount importance in striking the right balance between flexibility of adaptation and sufficient performance. To this end, some important design principles can be identified.

The specification of generic customizable implementations of common protocol functions organized in software libraries eases the burden of protocol stack software development through the reuse of prefabricated (and validated) functional components. This custom-off-the-shelf approach can also serve as an instrument for performance improvements of the protocol stack by selectively introducing components tailored and optimized for specific settings and functionalities. Furthermore, by leveraging object-orientation in design, unanticipated variants of protocol functionality can be accommodated and included in software libraries with relative ease.

The ability to dynamically adapt the protocol stack during runtime is determined by the degree of decoupling in the interactions between adaptation objects (e.g., protocol layers, protocol modules) which, in turn, is determined by the software architecture of the protocol stack. The dynamic switching between different implementations (e.g., different variants of the same protocol) during runtime is enabled by the protocol stack’s execution environment support for dynamic loading/unloading of protocol software code as well as by its software architecture’s capacity to enforce a quiescent state to affected software components.

Performance is a function of the allocation flexibility offered by the protocol stack’s processing model and the minimization of context switches occurring in its operation. In this respect, cut-through techniques in protocol layer processing and the collocation of interacting protocol objects in middleware approaches are of great value in boosting the performance of the protocol stack. As a final statement, although flexibility and performance lie on opposite sides of the design scale, a well-thought design that pays attention to the details, can strike the right balance, achieving great adaptability with negligible impact on performance.
Vangelis Gazis (gazis@di.uoa.gr) holds B.Sc., M.Sc., and Ph.D. degrees from the Dept. of Informatics and Telecommunications at the University of Athens, and an M.B.A. degree from the Athens University of Economics and Business. He is a senior researcher at the Communication Networks Laboratory (CNL) that he also Directs. He has conducted research in the areas of protocol design, performance evaluation, and software architectures, and has participated in national (OTE-DECT, GRNET, GUNET) and European IST projects (MOBIVAS, ANWIRE, MOBIVAS, ANWIRE, GEANT-2, SELFNET). His research interests include mobile service provision, reconfigurable mobile systems, cross-layer design, adaptable protocol stacks, and their dynamic reconfiguration.

Eleni Patouni (elenip@di.uoa.gr) received the B.Sc. and M.Sc. degrees (with honours) from the Dept. of Informatics and Telecommunications, University of Athens, Athens, Greece, in 2003 and 2005 respectively. She has been involved in the European IST-FP6 Integrated Projects E2R, E2R II and IST-FP7 E3 also leading WP4.1 "Self-management in autonomous system entities". She has over 20 publications in the thematic areas of beyond 3G, cognitive networks and autonomic communication systems and has received the best paper award in the IFIP Autonomic Networking Conference, 2006. She has also contributed in Open Mobile Alliance a definition of OMA Management Objects. Since 2006, she is pursuing her Ph.D. in the above thematic areas.

Nancy Alonistioti (nancy@di.uoa.gr) has a B.Sc. degree and a PhD degree in Informatics and Telecommunications (Dept. of Informatics and Telecommunications, University of Athens). She has worked as a project and technical manager. She has participated in several national and European projects, (CTS, SS#7, ACTS RAINBOW, EURESCOM, MOBIVAS, ANWIRE, E2R, LIAISON, E3, SELFNET, etc.). She has served as a lecturer in University of Piraeus and she has recently joined the faculty of Dept. Informatics and Telecommunications of University of Athens. She has over 70 publications in the area of mobile communications, reconfigurable, cognitive and autonomic systems and networks and Future Internet.

Lazaros Merakos (merakos@di.uoa.gr) is a Professor in the Dept. of Informatics and Telecommunications at the University of Athens. He is also Director of the associated Communication Networks Laboratory (CNL) research group and the Networks Operations Center (NOC) for the University of Athens. Since 1995, he has been leading CNL research activities in mobile communications in the framework of Greek and European Union projects under the Advanced Communication Technologies and Services (RAINBOW, MAGIC WAND) and Information Society Technologies (RAINBOW, MAGIC WAND, WINE, MOBIVAS, POLOS, ANWIRE, E2R, SCIER, SELFNET) programs. His research interests are in the design and performance analysis of broadband networks, and wireless/mobile communication systems and services. He has authored more than 180 papers in the above areas.