

A Protocol Framework for a Service-Oriented Future Internet Architecture

Bernd Reuther, Dennis Schwerdel, Abbas Siddiqui, Zornitsa Dimitrova and Paul Müller

*Integrated Communication Systems Lab
University of Kaiserslautern*

{reuther, schwerdel, siddiqui, dimitrova, pmueller}@cs.uni-kl.de

I. MOTIVATION

The current Internet architecture has some problems that could not be foreseen when it was invented, decades ago. A big problem is that the architecture itself cannot be changed in an evolutionary way. To overcome these problems we propose a novel architecture that solves all current problems and allows for further evolutionary development.

A. Problems within the current architecture

The strict layering of the current Internet architecture prevents protocols of different layers from communicating with and from using each other, which spawns several problems (e.g. the SSL part of HTTPS must provide a domain-specific certificate before the domain is transferred via the HTTP part.)

Another problem is that protocol headers are contained in other headers. That means that the outer envelope header has to be processed first, before the inner header can be processed which makes protocol processing a sequential task and also forces layering.

The contained inner protocol is identified by a header field of the outer header. Thus the set of possible inner protocols must be defined by the outer protocol in advance. Protocol selection is hard-coded in the applications, i.e. the application decides which protocol to use and therefore the developer has to know during development which protocols exist on the computer the application should run on. That makes it nearly impossible to introduce new protocols.

B. Our vision

We want to develop an architecture that allows to easily add and exchange protocols. To create such an architecture we identified three important changes to the current architecture.

Decoupling of protocols: Direct dependencies between protocols must be eliminated. That means that no protocol should require a specific protocol or contain another protocol header inside its own. Though, indirect dependencies between protocols may still exist. Protocols can require functionality („effects”) provided by other protocols, can consume data that is produced by other protocols or can emit notifications that are received by other protocols.

Flexible protocol composition: Protocols are self-descriptive, i.e. they have associated meta-data that describes which headers the protocol uses, which notifications it emits/receives, what dependencies it has and what effects it has. With this information, protocols can be combined to valid protocol graphs.

The protocol description also allows to calculate the compound qualitative and quantitative values of combined protocol graphs.

Service-oriented API: Applications no longer select the protocols, they describe their needs and let an agent select and compose protocols to fulfill those needs. Additional protocols can be dynamically included and excluded to improve overall performance.

C. Improvements over current Internet

With such an architecture it will be very easy to include new protocols. A new protocol is merely a network plugin that will automatically be used. Communication between protocols of different former layers will be possible and can improve performance.

The new architecture allows for fine grained micro-protocols, so for a given functionality a multitude of choices will exist. An agent selecting protocols from a big protocol pool will be able to offer new and significantly improved functionality for applications and adapt better to network constraints.

II. REALIZATION STEPS

The development of such an architecture can be split into three steps.

Step 1: Framework for operating loosely coupled protocols

The first step is to develop a framework in which protocols cooperate to process a data stream. The framework manages the processing of a work-flow, which is provided by the protocol selection and composition (Step 2) and provides mechanisms for data exchange avoiding direct interaction between protocols.

Step 2: Protocol selection and composition

The second step is to create a language for descriptions of protocols and based on these self-descriptions select and combine protocols, generating a valid protocol graph that fulfills the needs of the application.

Another task of that step is to create a service-oriented API for applications to express their needs for a combined protocol graph. Finally an agent will tweak and add protocols to improve the overall performance of the protocol graph.

In this step there are still some open issues. It is not decided how protocols are to be selected, if that could be done with optimization algorithms or with heuristics and precomposed meta-protocols.

Step 3: Heterogeneity

The third step is to handle a network of nodes with heterogeneous protocol pools. A path between the end-points must be found and established.

This step is currently an open issue. How could such a path be found? Possible solutions might include

- Translation between different protocols
- Minimal required base functionality in every node
- Transmitting protocol code to nodes

III. APPROACH

Our approach currently focuses on step one. In today's networks the work-flow and the data-flow between protocols is defined by the concept of layering. Layering implies sequential processing of protocols (layer N prior or after layer N+1) and also all data is passed from protocol to protocol. This process is controlled only by protocol identifiers (intermediate and receiver nodes) or by protocol selection (sender node). The goal for step one is to enable more flexible interaction between protocols using a layer-less approach.

Data flow: First there are data-flows between instances of the same protocols between different nodes. Because two protocols should be independent of each other, their control data should also be independent. So each protocol has its own headers using a type-length-value (TLV) encoding. Thus communication between nodes is represented as a sequence of TLVs (or Messages). The framework contains elements to distribute incoming messages to the protocols and to collect and forward outgoing messages.

Second there is data exchange between different protocols within a node. Instead of passing complete PDUs between protocols, each protocol defines which data it requires for input and which data it will provide as output. These data will be stored in a common data map („tuple space”). The tuple space is similar to today's protocol control blocks (PCB), but each value is accessed by well defined identifiers and the tuple space is shared between all protocols processing the same flow. So the tuple space enables data exchange between arbitrary protocols. In addition protocols store persistent internal states in the tuple space making the protocol implementations stateless.

Work-Flow: The framework provides dispatchers to organize the work-flow. They handle registrations of protocols for messages (TLVs). When a new header arrives, the dispatcher determines which protocols to run and their order. The default order in which protocols are called is defined by the order of messages, so no additional control data has to be transmitted.

Protocols can manipulate the work-flow to handle special situations (e.g. error behavior). Such manipulations include spawning other process threads, waiting for other threads to finish or to decide which protocol to call next. This allows for parallel header processing when the protocol dependencies permit.

In order to spawn new threads the dispatcher also manages notifications, which are internal generated messages enabling dynamic modification of the work-flow. Protocols can subscribe to notification types and emit notifications. The tuple space and notifications allow communication between protocols without having direct dependencies.

Some protocols must access other messages, e.g. for encryption or signing. These protocols may encapsulate other messages and may provide them for further processing to the dispatcher (in case of decryption) or a protocol may signal the dispatcher which other messages it will process (e.g. for signing). In both cases the relation of one spanning protocol to other protocols will be defined during protocol composition. This information will be encoded into the TLV of the spanning protocol and will be passed to the dispatcher where appropriate.

Finally a work-flow depends on the role of a node. An intermediate node will process different protocols than a receiver node. A role can be determined by protocols, for example while processing address information a protocol may determine if the current node is a receiver or not. In addition nodes may have configured roles, for example a node may choose to inspect messages for security reasons instead of processing them.

Composition: We plan to describe the functionality of protocols as provided „effects”. Protocols can require other effects and maybe other protocols. With these descriptions it might be possible to select and combine protocols, but this step is still an open issue. Such a protocol composition must be performed by a sender node and might be performed by intermediate nodes, e.g. adding sophisticated flow-control information or exchanging address types.