Lexis: An optimization-based model for the evolution of protocol stacks

Saamer Akhshabi, Constantine Dovrolis
College of Computing, Georgia Tech

August 28, 2013

1 Motivation

Network protocol stacks typically follow a multi-layer hierarchical architecture. What determines the required number of layers, or the number of protocols at each layer? The space of applications, services and user expectations (at the top layer) and the space of elementary functions (at the bottom layer) are constantly evolving – how does this *dynamic environment* affect the organization of layered protocol stacks? What determines the *evolvability* of a protocol in the presence of changes? And how can we design *new protocols* that can effectively coexist, or even replace, older incumbents that are widely deployed?

Why is it important to investigate these questions now? We argue that the current level of knowledge about protocol stacks is similar to that of shipbuilding in ancient Egypt – Egyptians were good at building ships (or some form of floating vessels) by the 4th millennium BC. They were doing so however without knowing about buoyancy, Archimedes' principle or the key equations of hydrostatics or hydrodynamics. It is likely that they had some empirical understanding of these concepts, but not an organized system of knowledge that we usually associate with the science of naval architectures today. Without such a system of knowledge every new floating vessel would be viewed as a new "invention", comparisons between the properties of different ships would be mostly subjective, and questions about the limits and capabilities of different designs would be tractable only after the corresponding vessel was built.

Our research in this area asks some basic questions about protocol design, and it aims to investigate the role of layering, modularity, complexity, and robustness in network architectures. The over-arching theme, however, is *evolution* – protocols and protocol stacks are not designed to work in a static environment. Applications, services, user expectations, communication and computing technologies, as well as the underlying economics, are all in a constant state of flux. A protocol stack designed in 2013 should not just be optimal in terms of what is known in 2013; instead, it should be able to evolve, and more importantly, maintain or improve its features while evolving.

We are studying the previous abstract but fundamental questions using a new model that we have recently developed, referred to as *Lexis*. Lexis is formulated based on cost objectives, and so it may be more appropriate for capturing the design and evolution of technological (engineered) systems compared to strictly evolutionary models. The model is described in Section 3. This extended abstract does not include any results. This is ongoing work and we expect to present our first results at the ITAT workshop in December'13.

2 An overview of Lexis

Lexis[1] captures *a cost-minimizing hierarchical design process.* The input to the design process is an alphabet where letters correspond to elementary functions. The output of the design process is a layered DAG that shows how to construct a given set of *regular expressions*, which correspond to the final applications that the architecture should provide. The objective of the Lexis design process is to construct the desired regular expressions hierarchically, first constructing simpler (*i.e.,* shorter) regular expressions (protocol modules) at internal layers, and reusing those as much as possible. The incentive to reuse existing regular expressions comes from the cost formulation: each regular expression has a cost that is related to its

[1] Lexis means "word" in Greek.

length relative to the length of its constituent regular expressions. The overall objective is to minimize the total cost of the designed architecture.

In Lexis, each protocol has an economic value. Fundamentally, each protocol has a value (or utility, fitness) that largely determines the evolution of that protocol, its competition with other similar protocols, and its likelihood of replacement. The value of a protocol X in Lexis depends on two related factors: a) the location of X in the protocol stack (typically, protocols with a large number of upstream dependencies are more valuable), and b) the number of people that directly or indirectly make use of X. Additionally, each protocol in Lexis has an economic cost, which depends on the complexity of that protocol's functionality relative to the complexity of its inputs. We believe that the cost of a protocol is determined by its *internal complexity surplus, i.e.,* the complexity of the additional functionality it provides relative to the complexity of the functionality provided by its inputs. For example, the cost of TCP is arguably higher than the cost of UDP because of the additional functionality provided by the former.

To evaluate the predictive power of Lexis, we examine whether the model can produce the following qualitative properties of existing and extinct protocol stacks:

- The number of lower-layer protocols and of higher-layer protocols is typically higher than the number of protocols at the middle-layers. A specific instance of this observation is the "hourglass pattern" in the Internet protocol architecture.
- We expect that the frequency at which new protocols are created, or existing protocols need to evolve, is higher as we move closer to the top or bottom layers of the protocol stack, compared to the middle layers. This is related to the observed "ossification" of protocols such as TCP or IPv4 in the case of the Internet protocol architecture.
- It has been repeatedly observed that whenever a new functionality is needed, perhaps to support a wave of emerging applications or user expectations, the protocol or mechanism that eventually "wins" is that which provides the desired functionality in the simplest possible way, or at the minimum possible cost.

3 Model description

This section presents *Lexis* in more detail. This model captures *a cost-minimizing hierarchical design process.* The input to the design process is an alphabet, E, where letters correspond to elementary functions. The output of the design process is a layered DAG, G, that shows how to construct a given set of *regular expressions*, S, which correspond to the final applications that the architecture should provide. Regular expressions are a simple but powerful modeling tool as they can represent any finite automaton. The objective of the Lexis design process is to construct the desired regular expressions hierarchically, first constructing simpler (*i.e.,* shorter) regular expressions (protocol modules) at internal layers, and reusing those as much as possible. The incentive to reuse existing regular expressions comes from the cost formulation: each regular expression has a cost that is related to its length relative to the length of its constituent regular expressions. The overall objective is to minimize the total cost of the designed architecture.

Consider an *alphabet* $E = \{e_1, e_2, \ldots e_n\}$ of n characters. Each character can be thought of as an elementary function in terms of computation, storage, transmission, etc. Even the simplest protocol module would require the appropriate interconnection of several such elementary functions.

A protocol module corresponds to a *regular expression* w that consists of characters from E combined with the three regular expression operators: union ($\cup$), concatenation ($\cdot$), and Kleene's star (*), listed here in order of increasing precedence. The union operator captures that a module's function requires either of two underlying functions, while the concatenation operator captures that both functions are needed in a certain order (as usual, we omit the notation ($\cdot$) for the concatenation operator). The star operator reflects that a certain function can be repeated an arbitrary number of times in a computation loop. A rigorous (recursive) definition can be found in many textbooks and is omitted.

More generally, a regular expression w can be *constructed* from a set $I(w)$ of one or more shorter regular expressions, referred to as *input operands* of w, if w can be written as a function of only members of $I(w)$ using the operators $\{\cup, \cdot, ^\star\}$. For example, the regular expression $w = a \cdot b \cup a \cdot (c \cup d) \cdot a \cdot b$ can be constructed from the set of input operands $I(w) = \{a \cdot b, a, c \cup d\}$, but it can also be constructed from the more elementary inputs $I(w) = \{a, b, c, d\}$. The fact that two applications or protocols may share similar functionality (*e.g.,* the reliable transfer of a byte stream) can be modeled with regular expressions that share similar segments. Also, the length of a regular expression represents the *complexity* of the corresponding function; longer (but not just repetitive) regular expressions represent more complex functionality.

Next, we present a static optimization problem, where the set of words, S, and the alphabet, E, are time-invariant and the design process is executed once. This is followed by a dynamic optimization in which the design evolves to include new elementary functions (changes in the alphabet E) and new applications (changes in the final regular expressions S).

Static Design: In the static design problem, we are given the alphabet E and a set of regular expressions S that represent target applications. The objective of the design process is to construct an *architecture* G, *i.e.,* a layered directed acyclic graph of regular expressions such that: a) the bottom layer nodes are the letters in E, b) the nodes of G include *all* regular expressions in S, and c) each expression w in G can be constructed from its input operands $I(w)$, *i.e.,* the nodes of G that have an outgoing edge to w. Note that the top-layer nodes are always regular expressions in S, but the converse is not necessarily true; some target applications may be input operands for other regular expressions. Fig. 1(a) offers a simple illustrative example.

The cost of a regular expression w is defined based on the cost of its input operands. Specifically, the cost $\gamma(w, I(w))$ of a regular expression w with input operands $I(w)$ is the *minimum* number of operators that are required to construct w from $I(w)$. It follows that the cost of a single-letter regular expression (*i.e.,* a regular expression at the bottom layer of G) is zero. For example, the cost of $w = ab \cup a(c \cup d)ab$ with $I(w) = \{ab, a, c \cup d\}$ is 3, while the cost of the same w using the input operands $I(w) = \{a, b, c, d\}$ is 6. In other words, the cost of a regular expression is defined based on the additional complexity (length) of that expression relative to the complexity of its input operands.

The following example illustrates an important point about the star operator. The cost of $w = ababab$ using $I(w) = \{(ab)^\star\}$ is zero because w can be constructed from the input operand $(ab)^\star$ without any additional operators. On the other hand, constructing w from $I(w) = \{a, b\}$ has a cost of two. In other words, the star operator allows us to construct a repetitive function at a lower cost.

Finally, the cost $\gamma(G)$ of an architecture G is defined as the cumulative cost of all regular expressions in G,

$$\gamma(G) = \sum_{w \in G} \gamma(w, I(w)) \tag{1}$$

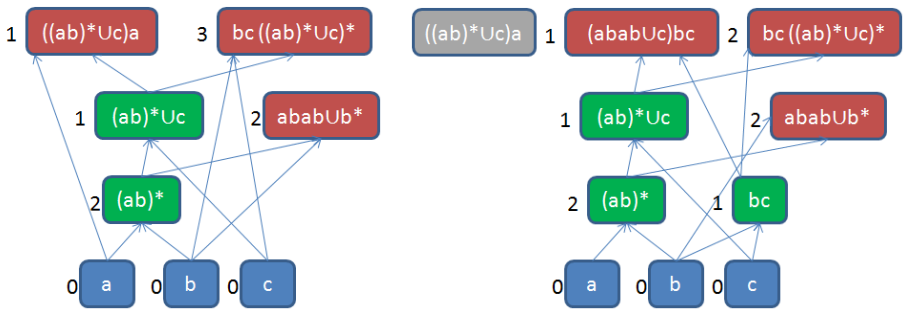
Note that this cost formulation provides the incentive to construct regular expressions located at internal layers of G, if they can be used as input operands for longer regular expressions. Additionally, reusing more complex (*i.e.,* longer) regular expressions results in lower cost than reusing simpler (*i.e.,* shorter) expressions,

The *static* hierarchical design problem is then to design a minimum cost architecture G given E and S: *Given a set of services S and an alphabet E, design a DAG architecture G that can construct all regular expressions in S and that minimizes the cost $\gamma(G)$.* A first important question is whether the problem in NP-Hard, and if so, are there efficient ways to solve it heuristically? Can we derive a good approximation bound for those heuristics? This algorithmic investigation will also focus on the factors that determine the optimal number of layers in an protocol architecture. A related question is: how does service *diversity* (in terms of entropy in the regular expressions that correspond to the given applications set) affect the resulting protocol stack? For instance, compare a network that offers just few services (*e.g.,* POTS) with one that has to offer many heterogenous services (Internet). Which of the two architectures will have more layers, and which architecture will have more intermediate protocols at each layer?

Dynamic Design: Suppose now that both the set S and the alphabet E can change with time, forcing the architecture G to also evolve. We want to understand how the dynamics of $S(t)$ and $E(t)$ can influence the architecture $G(t)$. Additionally, we want to understand how to design the architecture $G(t)$ so that it can be *evolvable*, *i.e.,* capable of evolving over a long time horizon at a low cost.

One approach is to redesign the architecture "from scratch" whenever there is a change in E or S, solving the static design problem after each change. This approach can be viewed as "clean slate" design, producing an optimized architecture at each point in time. It may have a significant cost overhead, however, because it does not attempt to reuse any of the existing protocols/modules (represented with internal regular expressions).

Another approach, is to incrementally design a new architecture $G(t+1)$ from an existing architecture $G(t)$, minimizing the *modification cost* $\gamma_\delta\left(G(t), G(t+1)\right)$ between the two architectures, *i.e., the cost of the new regular expressions (nodes) in $G(t+1)$ that are not present in $G(t)$*. This approach can be viewed as an *evolutionary design process*. It is important to note that even though this approach attempts to minimize the cost of each architectural transition, it may gradually produce architectures that are far from optimal, *i.e.,* with a total cost $\gamma(G(t))$ that is much larger than the cost of the optimal architecture at time t. A key question we plan to investigate is: *How does the evolutionary (incremental) design process compare to the optimized (clean slate) design process in terms of the cost of the overall architecture?* Is it that the cost of the former gradually deviates from the latter, leading to an increasingly bad architecture? Or is it that, depending on the dynamics of $S(t)$, the evolutionary design cost remains close to the optimized design cost?



(a) Initial architecture (b) Evolved architecture

Figure 1: **Left:** The alphabet E is shown in blue. The set of applications is shown in red. The green nodes represent regular expressions that are constructed as intermediate modules. The cost of each node is also shown. **Right:** The architecture evolves to support a different set of applications S' (the grey node is a removed application). A new internal node has been constructed. The modification cost is γ_δ=2 because of the two new nodes. The total cost of the architecture ($\gamma(G)$=9) has not increased however, because it is now "cheaper" to construct the existing applications.

Note that the evolutionary design process may both add and remove edges from $G(t)$. Removing edges can produce nodes without any outgoing edges. Such regular expressions are no longer required and can be removed from the architecture; this mechanism captures the removal of protocols from an architecture when they are not used any more (Property P8). The removal of a regular expression does not affect the modification cost.

We now state the evolutionary design problem: *Given an architecture $G(t)$ that constructs a set of applications $S(t)$ from an alphabet $E(t)$, and given a new set of applications $S(t+1)$ and potentially a new alphabet $E(t+1)$, design the architecture $G(t+1)$ that can construct $S(t+1)$ minimizing the modification cost $\gamma_\delta\left(G(t), G(t+1)\right)$.* The previous formulation implies a *myopic and memoryless approach* to evolutionary design. In other words, it does not try to learn from experience (how did $S(t)$ change in the past?) and it does not try to predict the future beyond time $t+1$. We will also consider extensions along those lines with more sophisticated evolutionary models, even though the design of most networking protocol stacks has arguably been rather myopic in practice.

Obviously, the dynamics in the time series $S(t)$ are very important in shaping the evolution of a protocol stack. A key question is whether the process that creates this time series is exogenous to the architecture $G(t)$ or not. New applications that can be easily constructed using the existing architecture may be more likely than those requiring completely new underlying modules and protocols. For example, it is easier to create an Internet telephony application (like Skype) that is based on the best-effort capabilities provided by IP and UDP than to create a telephony application that requires QoS guarantees. *How do the dynamics of $S(t)$ affect the resulting architecture $G(t)$?* We will start this investigation comparing the following two processes: a) each new application is generated independently of existing applications (exogenous creation of new regular expressions in $S(t)$), and b) each new application is a modification of one or more existing application (endogenous creation of $S(t)$). In both cases, services can become gradually more complex as captured by regular expressions of increasing length.

Finally, to evaluate the *robustness* of different architectures, we will also consider "disruptions" that may cause any of the following: a) introduction of new capabilities (new letters in the alphabet), b) introduction of completely new applications (regular expressions that have very low similarity with any existing expressions in $S(t)$), and c) sudden removal of many existing applications when a new "wave" of similar applications appears. *Which features make a protocol stack robust to such unexpected disruptions?* Another important question is when and whether it is simpler (*i.e.,* lower cost) to address such disruptions with a completely new architecture versus with incremental modifications.