On The Hourglass Model, The End-to-End Principle and Deployment Scalability

Micah Beck
University of Tennessee
Knoxville, TN 37996
mbeck@utk.edu

ABSTRACT

The hourglass model is widely used as a means of describing the design of the Internet, and can be found in the introduction of many modern textbooks. We argue that it also applies to the design of other successful shared interfaces, notably the Unix operating system kernel interface. The impressive success of the Internet has led to a wider interest in applying the hourglass design in other layered systems, with the goal of achieving similar results. However, application of the hourglass model has often led to controversy, perhaps in part because the language in which it has been expressed has been informal, and arguments for its validity have not been precise. Formalizing such an argument[1] is the goal of this paper.

CCS Concepts

•Networks → Layering; •Software and its engineering → Layered systems;

Keywords

ACM proceedings; LaTeX; text tagging

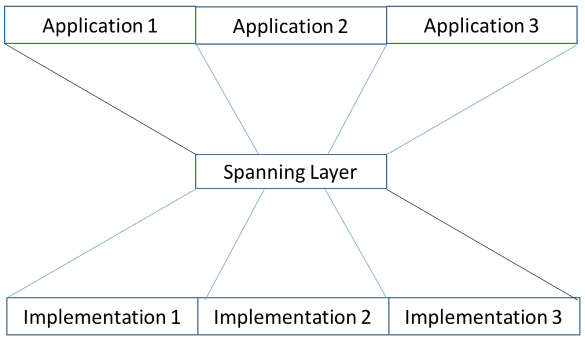

Figure 1: The hourglass model

1. INTRODUCTION

The hourglass model of layered systems architecture is a visual and conceptual representation of an approach to design in layered systems that seek to support a great diversity of applications and to admit a great diversity of implementations. At the center of the hourglass model is a distinguished layer in a stack of abstractions that is chosen as the sole means of accessing the lower level resources of the system. This distinguished layer can be given implementations using abstractions that are thought of as lying *below* it in the stack. The distinguished layer can be used to implement other services and applications that are thought of as lying *above* it. However, the components that lie above the distinguished layer cannot make direct access to the services that lie below it. The distinguished layer was called the "spanning layer" by Clark because in the Internet architecture it bridges the multiple local area network implementations that lie below it in the stack (see Figure 1) [5].

The shape suggested by the hourglass model expresses the goal that the spanning layer should support many diverse applications and have many possible implementations. Referring to the hourglass as a design tool also expresses the intuition that restricting the functionality of the spanning layer is instrumental in achieving these goals. These elements of the model are combined visually in the form of an hourglass shape, with the "thin waist" of the hourglass representing the restricted spanning layer, and its large upper and lower bells representing the multiplicity of applications and implementations, respectively (see Figure 1).

The hourglass model is widely used in describing the design of the Internet, and can be found in the introduction of many modern networking textbooks. A similar principle has also been applied to the design of other successful spanning layers, notably the Unix operating system kernel interface, by which we mean the primitive system calls and the interactions between user processes and the kernel prescribed by standard manual pages [9]. The impressive success of the Internet has led to a wider interest in applying the hourglass model in other layered systems, with the goal of achieving similar results [8]. However, application of the hourglass model has often led to controversy, perhaps in part because the language in which it has been expressed has been informal. The purpose of this paper is to present a formal model of layering and to use this model to prove some prop-

[1]An undertaking that was suggested to me many years ago by Alan Demers.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

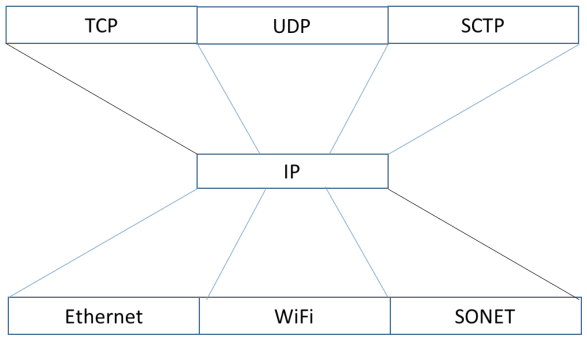

Figure 2: The Internet hourglass

erties relevant to the hourglass model. We will then use this model to explain the application of the hourglass model in the design of the Internet, and to show how it relates to other concepts in the design of layered systems, notably the End-to-End Principle.

2. OVERVIEW

We begin by presenting an abstract framework for reasoning about layered architectures and spanning layers in particular. We assume the existence of a certain relationship between layers, namely that one *layer specification* can *implement* another. We do not give a formal definitions for layer specifications or the *implements* relation here, as the complete formalization is somewhat complex and not, we believe, necessary to a satisfactory understanding of the argument. A more complete account of the formal framework is available elsewhere [1]. Given the existence of the *implements* relation, we then give definitions of *possible implementations* and *possible applications* of a layer in terms of it. Our account of the hourglass rests on two simple properties that can be derived from these definitions, and refer the interested reader elsewhere for formal proofs [1].

These definitions and the properties that we derive from them create a framework for characterizing a spanning layer in terms of the multiplicity of its possible applications and implementations. We explore the relationship between the multiplicity of possible applications and implementations and the logical strength or weakness of the spanning layer. We then use this framework to state and argue for the validity of The Deployment Scalability Tradeoff (DST).

> **The Deployment Scalability Tradeoff**
> There is an inherent tradeoff between the deployment scalability of a specification and the degree to which that specification is weak, simple, general and resource limited. n

The DST is intended to more formally account for the arguments made in classical papers that relate the hourglass design in layered architectures to the scalability of systems that they describe [10].

3. THE HOURGLASS

Definition 1. A *service specification* is a formal description of the syntax and necessary properties of a programming interface (API).

A service specification S describes an API: it specifies the behavior of certain program elements (functions or subprograms) through statements expressed in a programming logic. For instance, these might be such statements:

1. $\forall A, B \in Z\ [(A+1)+B = (A+B)+1]$
2. $\forall x, y \in N\ [\{x > 0\}\ y := x * x\ \{y > x\}]$

In formal terms a service specification is a theory of the programming logic. We denote by Σ the set of all such specifications expressed in the language of the specific logic. In practical terms, a service specification describes the operations of a protocol suite an or a programming interface such as operating system calls.

Definition 2. A specification $S1$ is *weaker* than another specification $S2$ iff $S2 \vdash S1$. $S1$ is *strictly weaker* than $S2$ iff $S2 \vdash S1$ but $S1 \not\vdash S2$.

Definition 3. An *implements* relation $S \prec_p T$ between two service specifications S and T and a program p expresses that in a model where API S is correctly instantiated, the program p correctly implements API T using the instantiation of S.

The *implements* relation is intended to be analogous to the "reduces to" relation of structural complexity theory.

3.1 The Hourglass Lemma

While we have omitted the detailed definitions of service specifications and the *implements* relation, we call upon the intuition of the reader to justify the following lemma which we present without proof. This lemma is the only place where the omitted basic definitions are used, and the remainder of our discussion is based upon the lemma.

LEMMA 1. *If $S1$ is weaker than $S2$, then*

1. $S1 \prec_p T \Rightarrow S2 \prec_p T$.
2. $T \prec_p S2 \Rightarrow T \prec_p S1$.

Proof omitted.

Both of these properites follow directly from fundamental definitions in program logic, and they also correspond very closely to covariance of return types and contravariance of argument types in object oriented inheritance [3].

3.2 Pre- and Postimages

We will express our formal analogs to scalability in terms of how large the sets of models are that can implement or can be implemented using a specification. To this end, we define the pre- and post-images of a specification under implementation.

These definitions are given relative to the set Π of programs that are considered as possible implementations of one layer in terms of another. We do not specify Π because we know of no accepted characterization of all "acceptable implementations" of one layer in terms of another. This is certainly a limited class, and is in fact finite since programs that are too large are considered unwieldy from a software engineering point of view. This class also changes over time, as hardware and software technology changes the set of capabilities that are available as implementation tools.

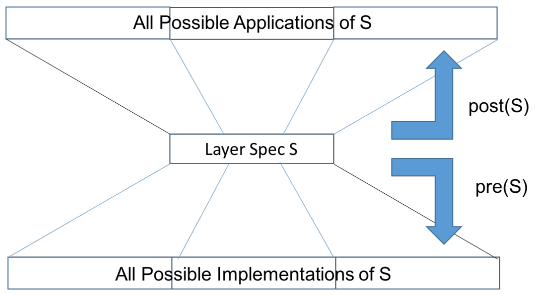
All Possible Applications of S
post(S)
Layer Spec S
pre(S)
All Possible Implementations of S

Figure 3: Pre- and postimages

Definition 4.

$$\mathrm{pre}_{\Pi}(S) = \{T \mid \exists p \in \Pi\ [T \prec_p S]\}$$

Definition 5.

$$\mathrm{post}_{\Pi}(S) = \{T \mid \exists p \in \Pi\ [S \prec_p T]\}$$

We acknowledge that in representing the set Π in our model as an external parameter we are not accounting for this aspect of the application of layered systems.

3.3 Using the hourglass as an analytical tool

Reference to the hourglass model is sometimes conflated with the idea of the spanning layer as a standard that is enforced by some external means such as legal regulation or as a condition of voluntary membership in some community. However, we can use our analysis of pre- and postimages of the *implements* relation as tools to analyze a layered system without necessarily relating it to any standards process.

If we select any set of services at one level of a layered system, we can ask what the design consequences would be if it were adopted as the spanning layer of a hypothetical system. Adoption as a spanning layer means that no other services would be available at that layer. Any participant in the system would have to use it as the sole means of accessing the services and resources of lower layers. Viewed in this way, the preimage of the *implements* relation denotes all possible implementations of the prospective spanning layer and the post image denotes all of its possible applications.

I use the term "denotes" because the pre- and post-image are not necessarily useful in actually enumerating these sets of specifications, since we have given neighter a formal specification for the value of Π, nor a way of determining whether a particular program p is in Π.

Taking a "descriptive" view of the hourglass allows us to use it as an analytical or predictive tool to understand the impact of a community's adopting a particular interface as a standard, be it de facto or de juris. Making the distinction between the use of the hourglass as a descriptive tool or as a means of justifying a standard also explains how different hourglasses can be examined and compared within the discussion of the same layered system. Every prospective spanning layer has an associated pre- and post-image, regardless of whether it is considered for any kind of standardization.

4. THE HOURGLASS THEOREM

This theorem is central to our understanding of the hourglass model.

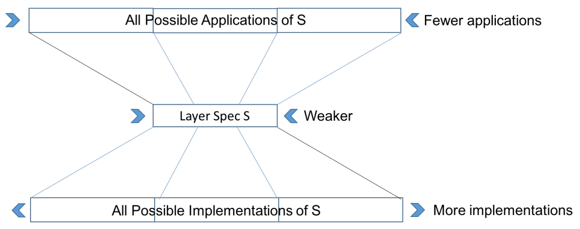
All Possible Applications of S
Fewer applications
Layer Spec S
Weaker
All Possible Implementations of S
More implementations

Figure 4: The Hourglass properties

Theorem 1. *If a specification S_1 is weaker than another specification S_2, then*

1. $post_{\Pi}(S_1) \subseteq post_{\Pi}(S_2)$

2. $pre_{\Pi}(S_1) \supseteq pre_{\Pi}(S_2)$

Proof:

1. By definition, $T \in post_{\Pi}(S_1)$ iff

- *$\exists p \in \Pi\ [S_1 \prec_p T\ ]$, so by Lemma 1*
- *$S_2 \prec_p T$, thus*
- *$T \in post(S_2)$*

2. By definition, $T \in pre_{\Pi}(S_2)$, iff

- *$\exists p \in \Pi\ [T \prec_p S_2\ ]$, so by Lemma 1*
- *$T \prec_p S_1$, thus*
- *$T \in pre(S_1)$*

$\square$

The Hourglass Theorem tells us (roughly) that a weaker layer specification has fewer applications but more implementations than a stronger one.

5. MINIMAL SUFFICIENCY

In terms of the hourglass shape, we can say that the "thin waist" (weak spanning layer) naturally tends to give rise to the "large lower bell" of the hourglass (many implementations). However a weaker spanning layer also tends to give rise to a "smaller upper bell" (fewer applications). Thus we must introduce some countervailing element into the model to ensure that it is in fact possible to implement all necessary applications.

We model, as a design goal, the necessity of implementing certain applications by introducing the set of necessary applications as another external parameter $N \subseteq \Sigma$.

Definition 6. A specification S is *sufficient* to implement a set of specifications N iff $N \subseteq \mathrm{post}_{\Pi}(S)$.

A spanning layer must be strong enough to implement all necessary applications but the stronger it is the fewer implementations are possible. We introduce the notion of minimal sufficiency as a means to balance these two design requirements:

Definition 7. A specification is *minimally sufficient* for N iff it is sufficient for N but there is no strictly weaker S' which is sufficient for N.

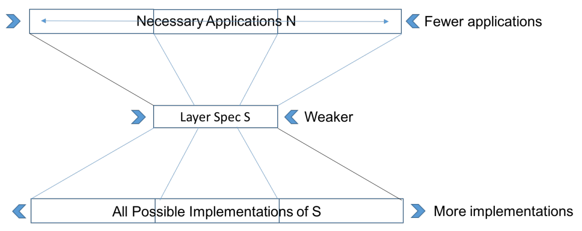

Figure 5: A minimally sufficient spanning layer

The balance between more applications and more implementations is achieved by specifying the set of necessary applications N and then seeking a spanning layer sufficient for N that is as weak as possible. This makes the choice of necessary applications N is the most directly consequential element in the process of defining a spanning layer that meets the goals of the hourglass model.

Note the implication that the trade-off between the weakness of the spanning layer and its sufficiency for a particular set of applications N is unavoidable. This suggests that attempts to design a spanning layer that both 1) achieves a high degree of weakness and also 2) is sufficient for a large set of necessary applications may have a tendency to fail.

6. THE END-TO-END PRINCIPLE

The End-to-End Principle is an idea that influenced the design of many layered systems, most famously the Internet. There has been much controversy associated with the End-to-End Principle, at least in part because it has often been cited as a fundamental reason for the success of the Internet, and so has aroused great passions in the network architecture community. In particular, this principle has been linked rhetorically to the "scalability" of the Internet, which is a somewhat ill-defined term we interpret in this paper as "having many implementations and many applications." We will now characterize the End-to-End Principle in terms of our model of layered systems, and explain how it may interact with the hourglass theorem while being quite distinct from it conceptually.

The End-to-End Principle as presented in the classic paper "End-to-End Arguments in System Design" by Saltzer, Reed and Clark [10] can be paraphrased as follows:

> In a layered architecture, any function should be located at the highest layer at which it can be correctly and completely implemented."

This principle is often justified by pointing out that different application communities, as represented in the layers of the stack that implement the functions they share, have different requirements, and these change over time. If we consider each layer of the stack to be the spanning layer of some community of interoperation then the lower the layer of implementation, the larger the community. If the interface to a given layer defines interoperability for a large community, it is difficult to address the various needs of that community, especially when it changes over time. Locating the function at a higher layer enables it to be specialized to the needs of a smaller community, and reduces the cost of changes to or replacement of its interface over time.

Viewed in this way, the End-to-End Principle is an exhortation to maintain abstraction and generality in the lower layers of the network, much as we do in the lower layers of any well designed object hierarchy. In any given example, this may result in more possible applications, not due to a lower layer having been strengthened but due to better maintenance of abstraction at the lower layers.

According to this analysis, the End-to-End Principle has no necessary relationship to the hourglass model.[2] If we compare a given specification S that implements a specific function to another layer S' that supports that function in a higher layer but does not implement it, S' may be weaker than S, it may be stronger, or the two may be incomparable. Illustrative examples given in Section 6.1.

However, in those cases where S' is weaker than S, the Hourglass Theorem tells us that S' may have more implementations. Thus, an application of the End-to-End Principle may (or may not) coincidentally result in greater scalability due to the effect described by the Hourglass Theorem. However, in those cases where the spanning layer can be substantially weakened in the process of moving a function to a higher layer, its impact can be two-fold:

1. the set of possible implementations may be increased due to weakening of the spanning layer, and simultaneously

2. the set of possible applications may be increased due to greater abstraction being maintained in the lower layers of the stack.

Such "heroic" applications of the End-to-End Principle in the design of the Internet may have given rise to a belief that it was the driving force behind the the hourglass model, leading to designs that maximize implementations. Our analysis suggests that this is only sometimes true, and that in some cases application of the End-to-End Principle may even lead to the strengthening of the spanning layer, and perhaps a reduction in the scalability attributable to the Hourglass Theorem! However, even in cases where weakening of the spanning layer is not possible, if the End-to-End principle can be applied while avoiding strengthening the spanning layer or while strengthening it only minimally, it may be possible to achieve both goals of the hourglass model (more applications and more implementations due to greater abstraction in the lower layers of the stack without strengthening the spanning layer).

6.1 End-to-End and Logical Weakness

Our analysis tells us that the End-to-End Principle has no necessary correlation with the logical strength of the spanning layer. To illustrate this point, we give one example where application of the End-to-End Principle results in a weakening of the spanning layer, and one in which its results in a strengthening.

6.1.1 Example: Reliable packet delivery

Consider a network layer that delivers packets from sender to receiver, and which requires that the sender annotate each flow of packets with a unique flow identifier and each packet with a unique packet sequence number. The network layer guarantees that packets in each flow are delivered in order and without missing sequence numbers. If such reliable delivery is not possible, the flow is terminated.

[2]This point was impressed upon me by Gerald Saltzer in his generous comments on an earlier version of this paper.

We can weaken this definition of the network layer by eliminating the significance of the flow identifier and sequence number, and allowing packets to be delivered in any order or to be dropped (as in IP). Flow identifiers and sequence numbers can be implemented at a higher layer (as in TCP). The weakened network layer will have more implementations, and reliability as implemented at the higher layer can be adapted to the needs of each application community.

6.1.2 *Example: Datagram timeouts*

Consider a best effort network layer that delivers datagrams from sender to receiver. Suppose we strengthen this network layer by adding an optional *timeout* field to the signature of the send function. The meaning of this field is that a datagram should not persist in the network for longer than the value of this field.

This would enable an application with hard real-time latency bounds to see only datagrams that are delivered within those bounds and to not waste network and endpoint resources on tardy ones. However, the strengthened network layer would have significantly fewer implementations, as it would require the use of accurate timers throughout the network layer.

7. OTHER ASPECTS OF THE THIN WAIST

Our analysis of the impact of the design of the spanning layer to application and implementation richness gives us a tool for making such a choice when considering the design of a layered system. If we agree on the limits of possible implementations (parameter Π) and the set of necessary applications (parameter N) we can then maximize the possible implementations of our spanning layer by choosing one that is minimally sufficient for N.

This analysis does not tell us how to design such a spanning layer, but it does give us an account of how certain choices may affect the adoption of any design. This model is however incomplete in that it leaves out many factors that have been considered important in the design of spanning layers. In an attempt to more fully account for prior discussions I will now relate some of those additional considerations to the formal model we have developed so far.

The logical strength or weakness of the spanning layer may be an appealing interpretation of the "thinness" of the spanning layer at the waist of the hourglass model largely because it yields to formalization using the tools of program logic. While this may account for some of the intention of prior references to the hourglass, it clearly does not capture those authors' intent entirely, since other factors have an impact on the value of a service definition as a potential community standard: simplicity, generality and resource limitation [10].

7.1 Simplicity

A requirement that is commonly given for the thin waist of the hourglass is that it should be simple. While logical weakness may be thought of as one aspect of simplicity, it clearly does not capture the entire concept. For example, one important aspect of simplicity that is not captured by logical weakness is orthogonality. In a service interface, orthogonality means that there is one way of gaining access to any fundamental underlying service or resource. Redundant features do not increase the strength of an interface but they do make it more complex. Software engineers understand the value of orthogonality in the design of interfaces

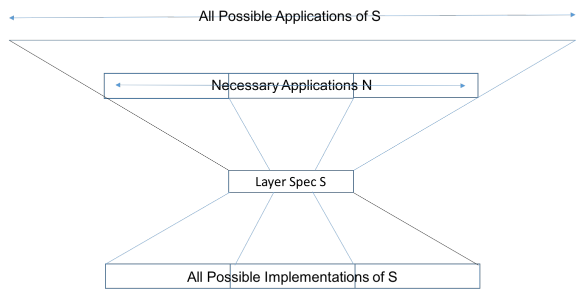
All Possible Applications of S
Necessary Applications N
Layer Spec S
All Possible Implementations of S

Figure 6: Generality

and are more likely to accept a design that has this form of simplicity as a community standard, but it is not accounted for in our formal discussion.

We understand simplicity as an important aspect of the acceptability of the spanning layer as a tool to be used by humans and in other contexts where resources may be limited or other factors may affect its adoption. If there are software engineering metrics or other formalisms that can capture these aspects of the design, then they could be incorporated into a more complete version of our model.

7.2 Generality

One unsatisfying aspect of our account of the "thinness" of the spanning layer as logical strength is that it does not account for the incredible diversity of applications that are supported by our two primary examples: the Internet and the Unix kernel interface. Our analysis implies that logical weakness of the spanning layer does not contribute to the diversity of applications, and in fact acts against it. So what accounts for the diversity we see in practice?

It is often observed that the diversity of applications supported by the Internet far outstrips those foreseen by its original designers. Thus we cannot say that the choice of necessary applications that went into the design directly determined the necessary strength of the spanning layer. (Perhaps the original designers are being modest, or had an implicit understanding of the eventual destiny of the network they were designing, but for the purposes of this discussion we will take them at their word.)

The power of both of these systems is related to orthogonality. Rather than crafting a spanning layer to directly support the functionality of the target applications they were considering, the designers crafted a set of orthogonal primitives such that all the target applications lay within the space of applications generated by them. Defining a set of orthogonal primitives is both an efficient strategy for implementing the set of target applications and also enables the implementation a highly diverse set of applications that are as yet unforeseen.

In terms of our model, the design of the spanning layer S yielded a very rich set of possible applications $\text{post}_{\Pi}(S)$. While the condition of sufficiency for a set of necessary applications is a more-or-less verifiable condition $N \subseteq \text{post}_{\Pi}(S)$, the set of all possible applications of a given spanning layer is much harder to evaluate, and designing a layer which tends to maximize it is an art.

Neither the Internet nor Unix would have had the impact they have achieved without simplicity and generality. One clue as to the origin of these imperatives within both the designers of the Internet Protocol and the Unix kernel interface may lie in a historical fact: Ken Thompson, Dennis Richie, Gerald Saltzer, David Clark and David Reed all participated in the Multics project, as did many of the prominent systems researchers of their generation [6].

Multics was an operating system project known for its many innovative features which had substantial success in reaching many of its technical goals, but which was also known for a high degree of complexity and lack of orthogonality. Multics is a classic example of a system that achieved its functionality goals but did not scale well. It is at least a reasonable hypothesis that these design imperatives, which have given us the most successful and scalable infrastructure interfaces in the history of computer systems — the Internet and the Unix operating system — were at least in part informed by the negative example of Multics in the areas of simplicity and generality.

7.3 Resource Limitation

The spanning layer provides an abstraction of the resources used in its implementation, preventing them from being accessed directly by applications. A such, it also defines the mechanism by which those resources are shared by applications and among users. In some communities, the modes of sharing are open, with few restrictions (e.g. resource quota) intended to ensure fairness among users. Such openness is one way of enabling the spanning layer to be logically weak (e.g. not implementing authorization of each user request). One way of managing more open modes of resource sharing is to limit the resources used by any individual service request, requiring large allocations of resources to be fragmented [7].

Such fragmentation allows for more fluidity in the allocation of resources (e.g. storage allocations), with competition between users occurring on a finer scale. Resource limitation means that use of the specification by an acceptable program will not result in overtaxing the resources of the platform on which it is implemented.

In other words, the thin waist of the hourglass is also a thin straw through which applications can draw upon the unprotected resources that are available in the lower layers of the stack. Resource limitation does not have a direct impact on the logical strength or weakness of the spanning layer, but it can affect the ability of the system to function in environments where there the demand for resources locally or transiently exceeds the capacity of the system.

8. DEPLOYMENT SCALABILITY

We have defined a model of a layered system of specifications and proved some properties relating the logical strength or weakness of one layer to the sets of possible applications and implementations. We then introduced the notion of a set of necessary applications as a design requirement of a spanning layer and defined some characteristics that seek to characterize the fitness of a specification in meeting that requirement. To augment this formal development, we have introduced three other ways of characterizing the "thinness" of a spanning layer: simplicity, generality and resource limitation.

We now seek to account for the idea that a system built on the a "thin" spanning layer is well adapted to finding success in the form of widespread adoption. We begin by giving a definition to this admittedly imprecise notion of success.

8.1 Deployment Scalability

Definition 8. We define *deployment scalability* as widespread acceptance, implementation and use of a service specification.

Deployment scalability is a problematic choice of goal because we have no clear way to specify whether or not it has been achieved. But as we are attempting to account for informal arguments, we may have to live with that. Our formal model depends on the parameters that a community may have trouble agreeing on: acceptable programs Π and a set of goal applications N. Then we have added three unformalized notions that we believe also influence the fitness of a spanning layer to achieve deployment scalability: simplicity, generality and resource limitation. Undaunted, we now offer a characterization of the tradeoff between these problematic elements.

8.2 The Deployment Scalability Tradeoff (DST)

> There is an inherent tradeoff between deployment scalability of a system with a given spanning layer and the weakness, simplicity, generality and resource limitation of that layer's specification.

The heart of the argument for the DST lies in the Hourglass Theorem, which explains why a layer that is minimally sufficient for N will maximize the possible implementations of that layer. Having the maximum possible choice of implementations is thus a key element of deployment scalability.

9. EXAMPLES AND APPLICATIONS

Giving full accounts of applications of the Deployment Scalability Tradeoff is a non-trivial matter, as the antecedents of the Hourglass Theorem require the definition of the specification language, a program logic, all acceptable programs Π and the set of necessary applications N. I will restrict my self to sketching some applications and giving a less formal account of the implications of the DST here.

9.1 Fault Detection in TCP/IP

The classic example of the application of the End-to-End Principle, from which its name is derived, is the location of the detection of data corruption or packet loss or reordering in the TCP/IP stack [10]. One argument for the location of the detection of such faults at the endpoints of communication (historically perhaps the original argument) is that it cannot be completely accomplished hop-by-hop because this does not account for errors that occur between hops, in the mechanisms and functioning of the intermediate nodes (routers). Our account of the hourglass model does not account for this argument, but models a different one.

The scalability argument for end-to-end detection of faults is that removing such functions from the spanning layer makes it weaker, and therefore potentially admits more possible implementations. Because fault detection can be implemented above the spanning layer, the set of applications supported is not reduced. So one point about referring to our model is that it enables a clear separation of the basis for two quite different arguments regarding the placement of fault detection, which might have otherwise been conflated because both are elements of the "thinness" of IP as the waist of the Internet hourglass.

Returning briefly to the argument that fault detection cannot be fully implemented hop-by-hop but can be implemented end-to-end, it is worth noting that it is less precise than the above argument based on logical weakness. In an end-to-end implementation of fault detection, there is still the possibility of error occurring within the implementation of TCP but outside the boundaries of the end-to-end check for errors. That is because sequence number and checksum verification occur within the mechanism of TCP, and there is some processing that occurs between those checks and the delivery of data to the application layer. Thus, while end-to-end checks reduce the locus of possible error from IP processing at every intermediate node plus all TCP processing to just a portion of the TCP processing at the endpoints, it does not in fact solve the problem completely in any formal or logical sense. I mention this difference not to disparage the effectiveness of TCP error detection, but simply to illustrate the difference between the application of the Hourglass Theorem, which is based on formal logic, and the argument regarding the incompleteness of hop-by-hop checking, which is a matter of reducing the opportunities for error.

9.2 Process Creation in Unix

In early operating systems it was common for the creation of a new process to be a privileged operation that could be invoked only from code running with supervisory privileges. There were multiple reasons for such caution, but one was that the power to allocate operating system resources that comprisee a new process was seen as too great to be delegated to the application level. Another reason was that the power of process creation (for example changing the identity under which the newly created process would run) was seen as too dangerous. This led to a situation in which command line interpretation was a near-immutable function of the operating system that could only be changed by the installation of new supervisory code modules, often a privilege open only to the vendor or system administrator.

In Unix, process creation was reduced to the `fork()` operation, a logically much weaker operation that did not allow any of the attributes of the child process to be determined by the parent, but instead required that the child inherit such attributes from the parent [9]. Operations that changed sensitive properties of a process were factored out into orthogonal calls such as `chown()` and `nice()`, which were fully or partially restricted to operating in supervisory mode; and `exec()` which was not so restricted but which was later extended with properties such as the *setuid* bit that were implemented as authenticated or protected features of the file system. The decision was made to allow the allocation of kernel resources by applications, leaving open the possibility of dynamic management of such allocation by the kernel at runtime, and creating the possibility of "Denial of Service" type attacks that persists to this day.

The result of this design was not only the ability to implement a variety of different command line interpreters as non-privileged user processes, leading to innovations and the introduction of powerful new language features, but also the flexible use of `fork()` as a tool in the design of multitasking applications. This design approach has led to the adaptation of Unix-like kernels to highly varied user interfaces (such as mobile devices) that were not within the original Unix design space.

As with the above example, reducing general process creation with `fork()` and `nice()` system calls can be seen as an application of the End-to-End Principle that also weakens the spanning layer (in this case the kernel system call interface). However, it should be noted that in some other ways the Unix kernel interface did strengthen the interface to the application layer, for instance by enforcing a strictly deterministic algorithm for allocation of file descriptors that enabled a careful calling sequence to be implemented between the command line interpreter and newly created processes. Thus, a simple application of the Hourglass Theorem may be instuctive, but the reality is somewhat more complex.

9.3 Replica Placement in Logistical Networking

Network storage virtualization has become an important component of distributed information technology resource management systems. Data replication and placement is often incorporated as a feature of the storage spanning layer that defines community interoperability in such systems but which is not under the explicit management of clients of that layer, accessible only through higher level abstractions. As a result, the policies that control such low level functions are either fixed or must be determined by clients through some policy interface of the virtualization layer.

The design of the Internet Backplane Protocol as the spanning layer of the Logistical Networking storage paradigm leaves the replication and placement of data to clients implementing higher layer functionality, such as distributed file systems or content distribution networks [2]. Operations that allocate storage and store data to or move data between storage intermediate nodes (sometimes called a Storage Object Target but referred to in Logistical Networking parlance as a "depot") are local to the depot to which they are directed. To facilitate the implementation of dynamic data movement, direct third party transfer between depots is supported.

This design enables diverse policy mechanisms to be conveniently implemented by clients of the storage virtualization service without interference from possibly inappropriate policies (e.g. cache coherence) being imposed in the implementation of the spanning layer. Clients that implement highly transient functions such as data streaming may decide to forgo replication, or to introduce it dynamically as a form of forward error correction only if network failure conditions are detected indicating that it would be efficacious. Clients implementing more persistent functions such as content delivery might use replication and data distribution much more aggressively in order to localize data throughout the network and to maximize the profitability of diverse multipath data downloading algorithms by end users.

This is another case where an application of the End-to-End Principle is accomplished through a weakening of the spanning layer.

9.4 Grid Authentication

In a retrospective lecture on "tussle spaces" in the design of networks, David Clark called out the lack of security at the Internet spanning layer as one regret regarding the design of the Internet. In today's difficult security environment, it is common to assume that some form of tight security is a necessity at the spanning layer, and in particular that authentication of identity should be a requirement of any use of common infrastructure.

The middleware framework for sharing of information technology resources that was given the communal name "The Grid" had strong authentication built in at the spanning layer of its protocol stack [8]. The Grid service stack was advertised as having a "thin waist" in analogy to the spanning layer of the Internet, seeking to lay claim to the implication of scalability. Grid authentication required that every user and resource under the management of the common middleware be assigned an X.509 Grid Certificate, obtainable only through a hierarchy of Certificate Authorities under the control of the U.S. Department of Energy or a similarly authoritative agency.

The exact impact of this requirement on the deployment scalability of the Grid is open to debate, but there is no question that a spanning layer that did not make this requirement for all access to common services would have had a weaker waist which would have had more possible implementations. The issue of what part, if any, of the substantial storage, networking and computing resources that were foreseen as being under the management of strong Grid authentication could have been responsibly accessed without such authentication, and what the implication might have been for the deployment scalability of Grid middleware, is beyond the scope of this discussion.

9.5 Process Management in Programmable Networking

PlanetLab is a platform for the allocation and use of distributed information technology resources in the form of intermediate nodes running a modified Linux kernel [4]. PlanetLab nodes are located throughout the United States, Europe and in some other parts of the world. The "spanning layer" of the distributed community of PlanetLab users consists of the shell command line, Internet, standard network services (e.g. `scp`) with some extensions for "slice" management, and a Linux kernel modified to implement increased isolation of resource utilization between slices. Resources of the intermediate node are allocated by executing commands and running servers that service requests of their own client communities.

The NSF-sponsored Global Environment for Network Innovation (GENI) also had with ambitious plans to provide a scalable network virtualization platform, and succeeding in some of those goals. Today, the inheritors of the mantle of network diversity lie in Software Defined Networking and Network Function Virtualization. Perhaps the hourglass can provide an analytical tool to help predict the likelihood that these approach will actually scale in deployment if their functionality is implemented in the spanning layer of a network or distributed system.

10. CONCLUSIONS

My interest in undestanding the hourglass and the End-to-End Principle grew out of the forcefully expressed assertion that integrating the use of storage into the Internet would somehow violate a fundamental rule and destroy its scalability. I have worked for over 15 years to understand how a necessary design feature of distributed system can be so dangerous as a feature of shared infrastruture. I have often been confused, and I have engaged in many confusing discussions and arguments, due I think in part to a lack of shared foundational defintions. It is my hope that this paper can shed some light on the history of the development of the most successful shared interfaces to have been developed in the history of the field of Computer Science, and to aid in the design of similarly successful systems in the future.

11. ACKNOWLEDGMENTS

I would like to thank the architects of all the systems that have motivated and informed my understanding of scalability in design, and the many mentors and colleagues who have encouraged and/or challenged my ideas.

12. REFERENCES

[1] M. Beck. On the hourglass model. *CoRR*, abs/1607.07183, 2016.

[2] M. Beck, T. Moore, and J. S. Plank. An end-to-end approach to globally scalable network storage. In *In ACM SIGCOMM 2002*, 2002.

[3] L. Cardelli. A semantics of multiple inheritance. In *Information and Computation*, pages 51–67. Springer-Verlag, 1988.

[4] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: An overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, July 2003.

[5] D. D. Clark. Interoperation, open interfaces and protocol architecture. In *The Unpredictable Certainty: White Paper*, pages 33–144. The National Academies Press, Washington, DC, 1997.

[6] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the multics system. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, AFIPS '65 (Fall, part I), pages 185–196, New York, NY, USA, 1965. ACM.

[7] G. Fagg, T. Moore, M. Beck, R. Wolski, J. S. Plank, A. Bassi, and M. Swany. The internet backplane protocol: A study in resource sharing. *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 00(undefined):194, 2002.

[8] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*, pages 47–48. Advanced computing. Computer systems design. Morgan Kaufmann Publishers, 1999.

[9] D. M. Ritchie and K. Thompson. The Unix time-sharing system. *Communications of the ACM*, 17:365–375, 1974.

[10] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, Nov. 1984.