Evolving beyond asynchrony

Tammo Spalink
Princeton University
TR-721-05

February 9, 2005

Abstract

Synchronous distributed systems deserve more research attention. This paper considers in more detail the advantages of wider adoption of synchrony, both for existing and potential novel applications. Supposing efficient and robust synchronization mechanisms existed, there are significant benefits to be had, even for existing applications. A technique is proposed to reduce or eliminate congestion for statistically multiplexed packet networks, using decentralized coordination of packet forwarding buffers. This technique can be implemented efficiently on a synchronous network because it leverages the available determinism – something missing from asynchronous networks. Formal system analysis techniques, such as temporal logic, assume synchrony because asynchrony creates too much non-determinism. Likely the main advantage of synchrony is enabling wider application of such powerful analysis, which is currently relegated to specialized domains such as safety-critical real-time control systems.

1 Introduction

This document argues for increased research effort in synchronous communication and in derivative distributed systems which allow deterministic resource control. Although synchronous communication currently used only for specialized systems, the recent development of an efficient algorithmic synchronous communication technique, enables its use in much wider contexts [22]. The implications of such use are explored for current infrastructures, in terms of potential new capabilities and improved performance. The concluded advantages are sufficient to motivate adoption of synchrony and deterministic resource control wherever possible, including systems such as the Internet. For example, an architecture is outlined below which leverages synchrony and determinism to eliminate traffic congestion for asynchronous packet communication, a significant Internet performance issue.

1.1 Vertical partitioning

For a distributed system which is shared by diverse and independent applications, interference between them in the form of contention for scarce resources can limit the usefulness of the system both in terms of performance and utilization.

For a sequential system, isolating applications from one another, to limit interference between them, is well understood by the operating systems community [20, 19]. In fact, some approaches such as Xen [3] and the Exokernel [9] are sufficiently strict to prevent interference entirely. These latter systems both provide abstractions that partition the resources of a physical platform into smaller virtual platforms with deterministic performance properties. What determinism means, in this case, is that applications can assume unconditional availability for a precise quantity of resources within their vertical partition during each discrete unit of time.

An alternative to this strict isolation is to employ a scheduler which distributes resources between applications in a more dynamic and unpredictable manner. Such a scheduler can be thought of as a layer of abstraction which hides properties of the real platform from applications. The reason for this might be, for example, to take advantage of the tradeoff between resource non-determinism and utilization.

Let the concept based on deterministic isolation be called *vertical partitioning*, to distinguish it from the horizontal layering approach. Certain applications simply require deterministic resource availability, meaning they are intolerant of temporal volatility in the performance properties of their implementation platform – streaming live media for example. In order to support these applications on a shared general-purpose infrastructure, it must provide vertical partitioning.

On distributes systems, as opposed to sequential systems, isolating applications from one another vertically is greatly complicated by physical distance and parallelism.

1.2 Synchronous communication

Given a network of processing nodes and communication links, the natural variation between the oscillators at each node forces all nodes to experience time independently. This means that there is no internal frame of time-reference which would allow any isolation technique to create determinism, unless the nodes are somehow synchronized. Even if all oscillators have a nominally equal operating frequency, they will only ever be "plesiochronous", or approximately synchronous in practice. This means that time could progress, as measured locally, faster or slower at different nodes.

However, this problem can be solved by imposing a strict order on the interaction between nodes, such that they can be synchronized to equally experience a discrete sequence of time steps. These steps can then be used to implement vertical partitioning.

To create this shared definition of time, the communication across all links in the network can be forced to occur in lock-step. If the communication is also forced to occur at a regular periodic rate, this frequency defines a shared notion of time for all nodes in the network independent of their individual oscillators. The technical details for a technique to achieve these communication properties robustly and efficiently can be found in [22].

Once a global communication frequency has been established for a distributed system, it becomes possible (in principle) to partition the resources in the system such that the quantity of available resources in each vertical partition is deterministic with respect to the shared time. For example, the processes in a distributed application can be coordinated across nodes to form a pipeline of computation and communication stages with precise total latency properties. In this sense, vertical partitioning is similar to the concept of "quality of service" (QoS) in networking research, although the latter is usually used to describe a more flexible range of resource availability guarantees.

1.3 Motivating examples

To achieve economies of scale, infrastructures are designed to support sharing of resources between multiple applications. Imagine dedicated independent platforms for each possible application in a complex system, and then recursively for each independent component within each application. This is obviously not a scalable approach. Instead, infrastructures are normally designed to be general-purpose, to support as broad a set of applications as possible. Maximizing the scope of what applications are supported by a platform best leverages the resulting economies of scale for performance, reliability, and operations.

The Internet is the most widely available, shared, general-purpose infrastructure. However, there exist applications which cannot be efficiently or safely deployed over the Internet. Instead, if they are not provisioned with dedicated platforms (share nothing), they only use a smaller fraction of the shared infrastructure such as lower level MPLS/ATM [21, 11] or SONET/SDH networks [23]. The value of the market for these applications is approximately US$28B [17]. In some cases sharing is actually impossible, for example, because the distributed system is mobile or physically disjoint from other infrastructure. However, in all other cases, reduced sharing implies increased cost.

Consider the following scenario as an example of sharing not practical using current infrastructure. Suppose that a safety-critical air traffic control (ATC) system were to share the same physical infrastructure with an interactive system for (possibly live) music distribution (such as iTunes) which is often targeted by (potentially global) flash crowds. Clearly the ATC system should be unaffected by activities of other applications on the shared platform, e.g. the release of popular information. If sharing between these applications were possible the cost of providing both services could likely be reduced. The degree of performance and robustness available for a given budget is likely higher than for any combination of dedicated platforms.

Alternatively, consider a hypothetical future application which requires deterministic resource sharing, not just for isolation from others others, but for simple correctness between its own components. Suppose a large number of geographically distributed people wish to virtually participate in a simulated sport or game. Further suppose that the quality of this experience depends on providing a virtual physics model with equivalent timing properties for all participants. Such an application would naturally require extensive reasoning about the temporal relationships between its processes, and hence between the resources of the implementation platform.

1.4 Overview

This document argues that most currently asynchronous infrastructures would benefit from deploying a synchronous "underlay", to make available extra capabilities. To this end, it will first formally define the necessary functionality. Once the desired new functionality is defined, the benefits for both existing asynchronous applications and for more novel or specialized applications are analyzed. Finally, the conclusion is preceded by a discussion of the implications for interoperability and integration with legacy systems.

The technical details for possible implementations are beyond the scope of this document, which only seeks to motivate additional research. However, related papers show that the required platform is not only possible, but practical and efficient across a wide range of platforms [22].

2 Synchronous multiplexing

As suggested by its name, synchronous multiplexing is the sharing of resources in a synchronous manner. Specifically, it means partitioning the resources in a system over time, for multiple disjoint flows of data, such that transitions across a sequence of resources can occur without intermediate buffering. While the common definition is specific to communication resources, think of this as a simple historical artifact – the concept applies equally well to other resources. If fact, it abstracts the mechanism required to implement vertical partitioning in a distributed system.

One example of a synchronously multiplexed system is a pipelined processor, where each stage in the pipeline is a resource, and a shared clock governs the duration for which each resource acts on a unit of data. For brevity, synchronous multiplexing will henceforth be called *synmux* in this document.

In the processor model, at each clock iteration, pipeline stages are all reused. To avoid being lost, the data at each stage must go somewhere at each time step. This property of regularity in resource usage is what makes synmuxed systems deterministic. For a pipeline of synmuxed resources, unless the pipeline becomes disconnected (failure may always cause data loss), the latency between any pair of stages can always be precisely calculated.

2.1 SONET/SDH

SONET [23], or the synchronous digital hierarchical (SDH) outside the US, is an example of an existing system based on the synmux principle. It is widely deployed, forming the basis for most of the global telecommunications backbone. However, few of the users of this system worry about performance or other risks of using this shared infrastructure as opposed to building a dedicated one – interference between users does not exist at this level.

In terms of widely deployed systems, SONET is unique in employing synmux. In brief, what SONET does is switch data between intersecting links at a global frequency of 8KHz. This means that, at each node, frames of data are received and transmitted for each link during every 125us time interval.This process only works if exactly the right amount of data is available at each node during each interval. If an upstream node were to transmit slowly and send less than the expected amount of data during an interval, demultiplexing might fail and result in communication failure. What is required is synchronous communication, meaning that the switching intervals at all nodes occur in lock-step.

There are two primary reasons why the SONET architecture cannot be easily generalized to multiple resources and other timescales. First, SONET achieves synchrony using the brute-force approach of equipping each nodes with a highly precise clock and coordinating them with an external clock synchronization system. Each clock must be sufficiently accurate and stable to have negligible drift between re-synchronization updates. Second, SONET is passive, meaning that reconfiguration occurs at human time-scales (minutes, hours) and not as the ongoing active result of computation – changes are coarse and costly. Thus, SONET provides only fixed bandwidth communication, and it makes for a poor general-purpose platform, being too specialized to suit the more dynamic needs of many applications.

2.2 End-to-end revisited

In their landmark paper, Saltzer et. al. collected and codified the community wisdom regarding the design of protocol layers in communication systems. The resulting "end-to-end argument" provides excellent design guidelines, not just for communication systems, but for managing abstractions in complex systems in general. However it does not address vertical partitioning, which allows a more general and thus more powerful end-to-end argument to be developed.

To manage complexity, systems are often horizontal partitioning into layers. Each layer is used to abstract the complexity of those beneath it, with the intent of simplifying the implementation of those layers above it. The end-to-end argument advises applying an imperative form of Occam's "razor" to this layering. The key of the argument says that each layer added to a system should add a minimum unit of functionality. This prevents a "pork-barrel" approach to layering, where higher level applications are forced to pay the cost of unnecessary features in monolithic lower layers. This philosophy is exemplified by the following quote from the end-to-end paper, which motivates elevating functionality as much as possible, but uses the term "level" instead of layer:

> "... performing the function at the lower level may cost more – for two reasons. First, since the lower level subsystem is common to many applications, those applications that do not need the function will pay for it anyway. Second, the low level subsystem may not have as much information as the higher levels, so it cannot do the job as efficiently."

The end-to-end guideline is paramount for a shared infrastructure. The Internet architecture, for example, imposes the Internet Protocol (IP) as the waistline for an hourglass of abstract interfaces. Below IP there can be many different platform-specific protocols, while above it are application-specific protocols. The capabilities of these higher levels are all bounded by those of IP. Thus, in general, any constraints imposed by such a "waistline" layer will naturally limit the scope of all higher layers.

Vertical partitioning, meaning the deterministic allocation of resources, can completely isolate the penalties of horizontal partitioning. Think of a vertical partition as similar in all respects to the underlying platform, but with less resources. By applying a stack of horizontal abstractions only within a vertical partition, applications which are incompatible with those abstractions can still be supported in other vertical partitions.

To summarize this reasoning, consider the following reinterpreted and extended end-to-end argument:

> When modularizing a system using horizontal layers, first provide an interface for vertical partitioning. Layers can then be isolated within a particular partition, and the generality of the system as a whole still preserved. To also maximize the generality of each partition, provide only a minimum of additional functionality with each layer. This allows higher levels the greatest flexibility to avoid unnecessary and potentially costly abstractions.

Often, horizontal layers impose a "policy" on the use of lower level resources. This can happen explicitly in the form of a layer implementing a scheduler which makes resource allocation decisions. It can also happen implicitly if the layer hides capabilities or properties available at lower levels. In this light, the new argument calls for systems to be designed in a policy-neutral manner, meaning that any layer which imposes a particular policy is isolated by a vertical partition.

This asymmetric need to allow vertical partitioning before imposing any horizontal constraints also highlights the similar lack of symmetry in the relationship between synchrony and asynchrony. Synchrony allows determinism which allows vertical partitions which can each support asynchrony, but not the reverse. This is why implementation of synmux must naturally occur at the low level. The ability to implement deterministic partitions is likely the primary advantage of synmux platforms, and the reason that they should be adopted as widely as possible.

3 Eliminating congestion

To illustrate the potential benefits of synchrony and determinism, this section describes an architecture to constrain contention for scarce resources in asynchronous packet communication, and thereby prevent congestion. Packets are the basic unit of communication in the Internet, and congestion is a significant source of inefficiency for that infrastructure.

Conceptually, the proposed solution has two parts. First, a conservative heuristic is used to prevent overflow for all packet buffers. This is done independently for each network link, using precisely timed feedback and leveraging deterministic latency and bandwidth. Applied to the whole network, the effect is to delay excess packets at their sources in overload situations. The second part of the solution utilizes lightweight communication circuits to provide short-cuts through the network for predictable traffic. These circuits maximize the effectiveness of packet buffers to handle unanticipated traffic. Together, these techniques should allow for full communication resource utilization without data loss, without retransmission, and with less processing and buffering resources than a similar (but lossy) asynchronous system.

3.1 Congestion model

When sending a packet in an asynchronous system, it is often impossible to know if resources will be available for it at the destination node, or along its transmission path. This is because non-zero communication latencies prevent globally shared up-to-date usage state. The widely employed *statistical multiplexing (statmux)* technique addresses this limitation by providing a waiting area for packets, in the form of dedicated buffer memory, to resolve temporary resource contention. So long as buffers are not full, packets can be forwarded between nodes without concern for precise link availability. If a link is temporarily full, packets can wait for it.

However, packets may be lost once buffers become full. If a packet arrives at a node, if the resource which it needs (outbound link) is unavailable, and if the packet buffers at that node are full, then the packet is simply lost (for it has no place to go). This state of events is generally called "congestion". The performance impact of lost data can be severe for some applications, since they must first establish that the data was lost and finally negotiate retransmission.

3.2 Reliable forwarding

A simple way to prevent buffer overflows is to apply flow control between all adjacent nodes. By providing feedback between nodes, they can inform one another about buffer availability and constrain data rates appropriately. However, the efficiency of flow control for two nodes depends on the consistency of the path between them. If path properties like bandwidth or latency can vary, the flow control can be only approximate or must be made sufficiently conservative tolerate any possible variation. Consequently, flow control can be implemented with maximal efficiently using a synmux platform with deterministic communication.

Consider a pair of directly connected nodes, identified as *source* and *sink* respectively, such that the source wishes to use some resource at the sink which is nondeterministically available. Assume that some memory is set aside at the sink to buffer data arriving from the source when the resource is unavailable.

Let a feedback message be sent at a regular interval I with the measured quantity of available buffer A_m at the time the message was sent. If the message is received at time T_{rx}, after a latency L, this means that A amount of buffer was available at time $T_{tx} = T_{rx} - L$. The source can then conservatively estimate the remaining available buffer A_{er}, by subtracting the amount of data which has arrived at the sink since T_{tx} from A. Note that this requires knowing the data transfer latency. By sending less than A_{er} data before performing the same calculation for the next feedback update, the source can maintain the invariant that data is not lost due to buffer overflow.

Basically, this protocol allows the source to send bursts of data to the sink. The size of a burst is bounded by the size of the buffer which the sink has dedicated to data arriving from the source. Once a burst worth of data has been sent, however, the source must wait for feedback. The source conservatively estimates that the resource may be unavailable and thus that the buffer may remain full (not drain). This estimate is then revised on each feedback iteration, based on actual measurements at the sink. Thus, the buffer capacity is never exceeded. When applied for all links in a network, these bounds on transmission achieve a form of distributed "source blocking", where nodes are unable to transmit more data than can be accepted by the network without risk of loss.

Because feedback is required to send more than one full burst of data, the sink has control over the source rate by choosing the buffer size. The maximum source transfer rate R (in data units per time unit) is determined by the following formula over the link latency L, the buffer size B, and the feedback interval I:

$$R = B \,/\, (\, L + I \,)$$

This equation can also derive the amount of buffer memory needed to support a specific source rate. However, since the relationship is not constant but a linear function, supporting high source rates can become expensive. This issue is addressed in more detail below, by providing an additional mechanism for large transfers.

3.3 Without determinism

On the other hand, if the nodes are not synchronized or if link latencies are nondeterministic, reliability becomes more expensive. Optimizing this process has received a great deal of research attention in the form of the congestion control technique used by the Internet transport control protocol (TCP) [18], but efficiency is limited by the great complexity involved. TCP congestion control is similar to flow control, but instead treats packet loss as approximate and indirect feedback. This is because there can be an indeterminate number of intermediate nodes along a communication path. While they may each cause variations in the path latency and bandwidth properties, direct communication with each one is impractical.

In the special case where the link latency is known and only synchronization is missing, interestingly, the flow control process itself can cause synchronization. This principle, combined with isochronous (fixed frequency) emptying of buffer memory, is the gist of an efficient process to provide network-wide communication synchrony [22].

The ultimate form of nondeterminism is physical component failure. If such failure occurs, data can always be lost. As explained by the end-to-end argument, tolerance for failure must always be dealt with at the application level, even if intermediate components are highly reliable.

3.4 Lightweight circuits

While careful flow control can make statmux forwarding reliable, it is also possible to configure synmux resources to create dedicated communication channels which are both reliable and more efficient.

Define *circuit* to indicate a fixed path communication channel between two nodes in the system with precisely known latency and bandwidth. Circuits are very simple to provide in a synmux system, since communication resources behave deterministically. Each link directly connecting two nodes is a circuit. If nodes are not directly connected, a circuit can be formed by combining those circuits along a path between them. This requires that intermediate nodes forward data between the adjacent circuits on the path. Assume that these connections can be rapidly created and destroyed at low cost, making the circuits *lightweight*.

The primary complexity with circuits is not implementing the circuits themselves, but rather the mechanism to establish and manage them. For example, the circuit network for voice telecommunications is very simple in principle – all channels have equal bandwidth (e.g. 64Kbps) and last for long periods of time (minutes). However, external networks of great complexity were built to operate it, such as TMN and SS7 [24, 25]. In turn, this lead to the misconception that circuit networks must be more complex or "smarter" (and thus more expensive) than packet networks.

For the purpose of the arguments being made here, it is sufficient to assume that some circuit allocation and management system exists. The implementation of the allocation system for circuits are beyond the scope of this document. Because the synmux platform assumed by this document includes processing and storage resources as well as communication, this allocator does not need to be an external system like TMN – it can exist within a vertical partition. This avoids a fundamental (often ignored) problem with external management systems, which is managing the management system. An example allocation system could be a vertical partition across the whole network, within which a subset of the Internet protocol suite is deployed. Circuit establishment could involve RSVP [2], modified to configure synmux resources instead of IP routers.

3.5 Traffic self-engineering

Neither of the above techniques perform well in isolation, but they seem to complement one another perfectly. The trouble with statmux, is that non-"bursty" traffic clogs up forwarding buffers. Continuous data flows imposes a constant memory load, reducing the burst-capacity for other incoming data. The trouble with circuits, on the other hand, is the cost of establishing them. The extra communication to configure circuit resources (signaling), imposes both latency and bandwidth overhead. However, these drawbacks effectively cancel one another out when the approaches are applied cooperatively.

Traffic engineering for a packet network is the process of provisioning capacity to match usage patterns [1]. The above techniques support this in a fully automated fashion. By combining rapidly configurable circuits with a congestion-free statmux network, short-cut circuits can be created for any traffic that would otherwise impose ongoing buffer utilization.

Let a network be initially configured such that all link bandwidth is used for statmux packet data, extended with the reliable forwarding feedback and associated dedicated buffer memory at each node. This facilitates low latency communication for small messages, where the maximum size is determined by the burst capacity of the buffers along the message path. Large messages are possible as well, but they may end up spread across the buffers on multiple nodes and suffer latency penalties accordingly.

Short-cut circuits are established when nodes face ongoing utilization. Consider a chains of three nodes *A*, *B* and *C*, connected in that sequence. If at any time, *A* is forwarding packets through *B* to *C* at some non-zero average rate, the buffer utilization at *B* could be improved by creation of a circuit to directly connect *A* and *C*. This is true for all such chains of nodes. In fact, *B* need not be just one node, it can represent a whole cloud that is best bypassed. This situation can be recognized at any *A* node by monitoring outbound traffic and identifying flows with overlapping paths. If *A* is the source for a multi-packet message, it may thus create a circuit for the full path to the destination and thus eliminate any statmux penalty for large messages.

Given some circuit signaling mechanism, the bandwidth for these short-cut circuits can be modulated over time to match any continuous or predictable traffic load. Using this technique, the latency benefits of a packet network are retained while eliminating congestion and improving utilization – all for arbitrary traffic patterns. It is not coincidental that circuits and statmux work so well together. Statmux is naturally a local allocation process, while circuits are a distributed one. Traffic that exceeds local resource availability cannot be addressed locally, rather a distributed process should ideally cause traffic to be generated at the source at an appropriate rate to match bottlenecks along its path. Circuits embody the distributed signaling and distributed state needed to implement this.

Although the details here may be novel, this use of circuits bears strong resemblance to previous work, including that by Zhang, Fernandez, and McKeown [17, 16] on using ATM virtual circuits for TCP traffic. An important difference, however, is that the technique here do not depend on any higher level protocol to provide connection information. Short-cuts can be created in a general manner for all higher level causes of ongoing resource contention.

4 Applied temporal logic

No less significant than improving existing applications, is the prospect of enabling new ones. The field of temporal logic provides powerful tools to analyze and design complex systems, but which generally assume synchronous communication. While it is possible to automatically translate synchronous systems into asynchronous ones, there is normally a cost in terms of communication overhead (application synchronization) or in terms of lost parallelism. Synmux platforms provide the needed synchrony, and hence can enable those applications that cannot tolerate the translation penalties.

Temporal logic is currently widely used for specifying and verifying the correctness of programs, especially concurrent ones. A special type of modal logic, it provides a formal system to reason about changes in the truth of assertions over time. Examples of temporal logic operators include *sometimes* and *always*, in addition to the traditional operators in boolean logic. Naturally, the logic assumes that a single discrete sequence of time applies to all program components.

4.1 Testing for absence

Possibly the most important feature of a synchronous system is the ability to test for the absence of data on a communication channel. This concept is fundamental to digital circuit design, since binary data is often defined as either the presence or absence of signal on a channel. Synchronous communication provides the same ability for distributed systems, the ability to know when to sample a channel without explicit coordination.

This is obviously impossible for an asynchronous system, since there is no time relationship between components by definition – a component would not know when or for how long to wait. A plesiochronous system on the other hand provides some information of how long to wait, but still requires additional synchronizing communication to establish when to test – effectively defeating the efficiency implied by "absence" of data.

The use of timeout mechanisms in architectures like the Internet is an interesting example. The timing intervals for timeouts is often extremely coarse, reflecting an assumption of rough synchrony for the system, at least at the level of human time (seconds). This assumption is often reasonable, because routers which can arbitrarily take either nanoseconds or years to switch a packet are effectively useless. However, greater synchrony would allow greater efficiency and robustness.

4.2 Reactive systems

Consider what kinds of programs are best targeted by powerful temporal logic analysis. Suppose programs are classified into two groups, those that are self-contained with sequential execution, and all the rest. Sequential self-contained programs are given starting inputs, and they simply perform calculations without further interaction until providing a result and terminating. It is the behavior of the other class of programs, called *reactive systems*, which is more complex and hence required powerful tools to help reason about [8, 14].

Reactive systems include those programs which are designed to continuously interact with their environment, meaning other programs, sensors, actuators, etc. These programs follow a "reactive cycle" of input, processing, and output. Because of this, they can even experience feedback, meaning that the outputs during one cycle may affect the inputs during future cycles. This class of programs is the usual focus of temporal logic. Examples of reactive programs include network protocols, process control systems, etc.

4.3 Synchronous languages

Esterel [7], Lustre [13], and Signal [12] form a family of languages designed to express reactive programs in a manner that promotes the application of temporal logic. Central to their design, is the assumption that all concurrent processes are synchronized to share a sequence of discrete time steps. This property allows the language to enforce deterministic interaction between software components and thereby increase confidence in overall system correctness. Accordingly, these languages have had commercial success in the domain of safety critical systems such as those for avionics, automotive control, and nuclear power plants [6].

To ensure the correctness of a system, both the software and the hardware require extensive analysis. While these languages provide capabilities to ensure software determinism, the lack of general purpose deterministic platforms often requires expensive custom implementations for their programs. Within the realm of hard-real-time systems, efforts such as the Time-Triggered Architecture [15] seek to address this need.

Increasing confidence in software correctness is always important, including for general purpose shared infrastructures. In hardware design, synchronization has long been the foundation for ensuring the correctness of complex systems. The synchronous language community has demonstrated that the same principles apply to software. However, the implementation platform for such software is missing.

Without a deterministic synchronous platform, implementation of synchronous programs must be "desynchronized" [5]. This means either that all communication between concurrent components must include explicit synchronization signaling, or that concurrency is eliminated by imposing a sequential total order on the program. Both solutions are undesirable. The former imposes significant overhead and prevents hard real-time applications which must remain not only internally synchronized, but also interact with devices in a timely manner. The latter eliminates any scalability otherwise available through parallelism.

5 Conclusion

Given that synchronous communication can be efficiently implemented on a wide variety of platforms, most computing and communication infrastructures should evolve to include this capability and exploit its benefits. This is not to say that asynchronous approaches should be superseded, simply that they are not universal. Instincts to assume that they are, on the part of the community, should be reconsidered. Synchronous architectures deserve of far more research attention. Because this domain is significantly outside the familiar subjects of the community, this document tries to extensively motivate its relevance, at the expense of technical details. Supporting technical papers provide specific design and implementation points to show that achieving the needed platform capabilities is not only possible, but practical.

A convenient feature of vertical partitioning is that it simplifies supporting legacy systems – they can simply be implemented in such a partition and possibly be none the wiser. Take the Internet architecture for example, or even better an extension of it to support computation and storage such as PlanetLab [4] or the Grid [10]. Any of these can be deployed within a distributed virtual partition with no negative performance implications. In fact, multiple instances of possibly different software versions could co-exist side by side.

While it might seem unrealistic to change core infrastructure in the fundamental manner needed to support synchrony and determinism, consider that the new platforms can be deployed and also taken advantage of incrementally. This is because it can easily interface with legacy systems, while only exposing the new functionality where needed.

References

[1] Daniel O. Awduche, Angela Chiu, Anwar Elwalid, Indra Widjaja, and XiPeng Xiao. Overview and principles of internet traffic engineering. Internet-Draft Version 02, Internet Engineering Task Force (IETF), 2001.

[2] F. Baker, B. Braden, S. Bradner, M. O‘Dell, A. Romanow, A. Weinrib, and L. Zhang. Resource reservation protocol (RSVP). Request for Comments (RFC) 2208, Internet Engineering Task Force (IETF), September 1997.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, Bolton Landing (Lake George), NY, October 2003.

[4] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale network services. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 253–266, San Francisco, CA, March 2004.

[5] Albert Benveniste, Benot Caillaud, and Paul Le Guernic. Compositionality in dataflow synchronous languages: specification and distributed code generation. *Information and Computation*, 163(1):125–171, November 2000.

[6] Albert Benveniste, Paul Caspi, Stephan A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

[7] Gerard Berry. The foundationis of esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. The MIT Press, 1998.

[8] E. Allen Emerson. Temporal and modal logic. In *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 995–1072. The MIT Press, Cambridge, MA, 1991.

[9] Dawson R. Engler and M. Frans Kaashoek. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, Copper Mountain Resort, CO, October 1997.

[10] Ian Foster and Carl Kesselman, editors. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1999.

[11] Jim Grace, Richard Breault, John Jaeger, and Lou Wojnaroski. ATM user-network interface specification v3.0. Standard Specification af-uni-0010.001, ATM Forum, September 1993.

[12] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. Signal – a data flow-oriented language for signal processing. *IEEE Transactions on Signal Processing*, 34(2):362–374, April 1986.

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 1991.

[14] Nicholas Halbwachs. Synchronous programming of reactive systems: a tutorial and commented bibliography. In *Proceedings of the International Conference on Computer Aided Verication (CAV)*, number 1497 in LNCS, pages 1–16, Vancouver, British Columbia, Canada, June 1998.

[15] Hermann Kopetz and Gunther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, January 2003.

[16] Pablo Molinero-Fernández and Nick McKeown. The performance of circuit switching in the internet. *OSA Journal of Optical Networking*, 2(4):83–96, March 2003.

[17] Pablo Molinero-Fernández, Nick McKeown, and Hui Zhang. Is IP going to take over the world (of communications)? In *ACM HotNets*, October 2002.

[18] Jon Postel. Transmission control protocol (TCP). Request for Comments (RFC) 793, Internet Engineering Task Force (IETF), September 1981.

[19] Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alesandro Forin, David Golub, and Michael B. Jones. Mach: a system software kernel. In *Proceedings of the IEEE Computer Society International Conference (COMPCON)*, pages 176–178, San Francisco, CA, March 1989.

[20] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Communications of the Association for Computing Machinery (CACM)*, 17(7):365–375, July 1974.

[21] E. Rosen. Multiprotocol label switching architecture. Request for Comments (RFC) 3031, Internet Engineering Task Force (IETF), January 2001.

[22] Tammo Spalink. Communication synchronization. Technical Report TR-722-05, Princeton University, 2005.

[23] Telcordia Technologies, Inc. Synchronous optical network (SONET) transport systems: Common generic criteria. Document Number GR-253, September 2000.

[24] Divakara K. Udupa. *TMN: Telecommunications Management Network*. The McGraw-Hill Companies, Inc., New York, NY, 1999.

[25] John G. van Bosse. *Signaling in Telecommunication Networks*. John Wiley & Sons, Inc., New York, NY, 1998.