A Promise Theory Perspective on Data Networks

Paul Borrill
EARTH Computing, Inc
Palo Alto, CA 94306
paul@borrill.com

Mark Burgess
CFEngine, Inc.
Mountain View, CA 94040
mark.burgess@cfengine.com

Todd Craw
Cumulus Networks
Mountain View, CA 94041
todd@cumulusnetworks.com

Mike Dvorkin
Cisco Inc.
San Jose, CA
Mike.Dvorkin@cisco.com

***Abstract*—Networking is undergoing a transformation throughout our industry. The shift from hardware driven products with ad hoc control to Software Defined Networks is now well underway. In this paper, we adopt the perspective of the Promise Theory to examine the current state of networking technologies so that we might see beyond specific technologies to principles for building flexible and scalable networks. Today's applications are increasingly distributed planet-wide in cloud-like hosting environments. Promise Theory's bottom-up modelling has been applied to server management for many years and lends itself to principles of self-healing, scalability and robustness.**

DRAFT April 26, 2014

I. Introduction

Modern networks have reached a point where some have begun to argue that is it difficult to abstract or manage them without returning to a centralized and imperative model [LWH+12]. Network design focuses on legacy data structures and protocols rather than the business functionality required from the network. A modern approach to network design emphasizing simplicity and relevant abstraction seems overdue. Such an approach could reduce the cost and brittleness of network design. A few leaders are already exploring some of these ideas, building highly-scalable, autonomous networks that behave in predictable ways.

The 'Promise Theory', was introduced in 2005 as an approach to modelling distributed systems based on complete decentralization [Bur05]. Coupled with abstraction, it offers a looking glass to simplify the design and management of networks. If we define what a user or application needs from the network we can begin to get away from imperatively controlling the 'how' the network functions and instead focus on declaratively describing "what" is required from it. A Promise network begins as one in which the network elements act as autonomous agents, and collaborate to find the best way to deliver the required function.

In this paper we wish to apply Promise Theory as a measuring stick for the state of the art in networking today, with the aim of clarifying the important promises needed to make network infrastructure work. This needs to be done in a way that is independent of the technologies used to keep them. In this way, we cast a critical eye over current practice and future directions. We are able to show that there are simple unifying principles for networking that are independent of scaling arguments, and that there is no need to base future networking on the confines of the traditional OSI model.

II. Promise Theory

Promise theory is a theory about what can happen in a collection of components that work together [Bur05], [MK05]. It starts from the 'bottom' with low level components and looks upwards to build cooperative structures. This is an unusual viewpoint for Computer Science. Rather than adopting the conventional belief that only that which is programmed happens, it takes the opposite viewpoint: "only that which is promised can be predicted". It therefore approaches management embracing uncertainty—one might say with realism rather than faith [Bur13].

Promise theory begins with the idea of completely autonomous agents that interact through the promises they make to one another. It is therefore particularly well-suited to modeling ad-hoc networks [MB04]. Although we cannot force autonomous agents to work together, we can observe when there are sufficient promises made to conclude that they are indeed cooperating in the keeping of a promise, initiated by one of them. The application of Promise Theory lends itself to describing policy-governed services, in a framework of completely decentralized or autonomous agents that assist one another by voluntary cooperation alone.

Our challenge in this paper, is to translate this bottom-up effectiveness of promise theory in ad-hoc networks, to the world of top-down, human managed networks which is the norm today, and see what insights can be drawn from the overlap of those requirements.

Agent is the term for are the fundamental entities in Promise Theory. Agents are not necessarily like 'software agents', they can be any active entities like an interface that keeps promises[1]. Actions taken by agents are not in the scope of Promise Theory. We assume that appropriate actions are taken to keep the promises. In that way, we focus on declarative intent, rather than imperative procedures.

A. Formalism

The promise formalism has a number of features, described in [BB14]. We refer readers to this reference for details.

[1] A perspective from physics would suggest the most important property of fundamental entities is that they be *upwardly heritable* (a phrase borrowed from Nobel laureate F. Wilczek in discussing the building blocks of nature [Wil06]). This requires microscopic laws that, when consistently applied to large bodies, *retain their character*. In networks, promise theory provides this notion of upwardly heritable. At the lowest level, we have agents, applied to the cell, molecule and atom of our model. With agents in each layer providing promises to each other and to the agents in the larger bodies above, that retain their character. In this way, we can expect desired behaviors of the infrastructure as a whole, such as self-healing, scalability, and robustness, to emerge.

A *promise* is an intention that has been autonomously adopted by an agent (the source of its agency is usually a human owner, or perhaps an agreed standardization). An agent that only promises to do as it's told is *dependent* or voluntarily subordinated. It has some of the characteristics of a service: an agent makes its intended behavior known to other agents (e.g. I will serve files on demand, or forward packets when I receive them). An imposition is an attempt to induce the cooperation of another agent by imposing upon it (e.g. give me the file, take this packet).

We write a promise from $\mathrm{Promiser}$ to $\mathrm{Promisee}$, with body b as follows:

$$\mathrm{Promiser} \xrightarrow{b} \mathrm{Promisee}. \tag{1}$$

and we denote an imposition by

$$\mathrm{Imposer} \xrightarrow{b}\!\!\blacksquare\ \mathrm{Imposee}. \tag{2}$$

Promises and impositions fall into two polarities, denoted by $\pm$. A promise to give or provide a behavior b is denoted by a body $+b$, which a promise to accept something is denoted $-b$ (or sometimes $U(b)$, meaning use-b). Similarly, an imposition on an agent to give something would have body $+b$, while an imposition to accept something has a body $-b$.

To complete a transaction, we need a match an imposition (+) with a promise to use (-). To form a binding (as part of a contract), we need to match a promise to give (+) with a promise to use (-). Without these bindings, autonomous agents can simply disregard one another's intentions. This discipline forces one to document necessary and sufficient conditions for cooperative behaviour.

A promise model thus consists of a graph of nodes and edges, where nodes are called *agents*, and edges can be interpreted as either *promises* or *impositions*. Agents communicate their intentions by making promises and impositions on other agents, and they self-determine their behavior based on this information. Promises and impositions are not a network protocol: whatever protocol might be used to communicate promises is not defined (and shouldn't be). Agents publish their intentions and other agents may or may not choose to pay attention.

Promise Theory is a way of breaking down any system into individual component parts, like a table of elements, which are bound together through abstract bonds called promises. In that sense, it forms a chemistry of intent [Bur13]. It has no particular manifesto, other than to decompose systems into the set of necessary and sufficient promises to model their behavior. However, the fact that this is possible indicates the possibility of a form of engineering which runs contrary to many learned practices in the IT industry.

III. Basic networking through the eye-glass of promises

Promise Theory allows us to discuss and reason about autonomously composed systems without getting bogged down in technology details. Let us examine the basics of present day networking using a promise viewpoint. The basic agents of networking today form the interfaces of the communications stack. A simple way to model these is the assume that there are 'virtual' interfaces for the different OSI layers. The main thing that distinguishes these layers from a formal viewpoint is the data format they use.

The basic agents in a data-networking model are the interfaces. These make various promises in order to be able to communicate data with one another, and forward data.

A. Ethernet (L2)

In the Ethernet protocol, interfaces promise to label transmissions with unique MAC addresses. A MAC address is a unique string of digits that promises to be unique. Thus each agent E_i, or interface, promises that its own MAC address is unique to every other:

$$E_i \xrightarrow{+\mathrm{MAC}_i|\mathrm{MAC}_i\neq\mathrm{MAC}_j} E_j \quad \forall i,j \tag{3}$$

$$\tag{4}$$

When data are transmitted by an interface, it promises to use messages of the form: (destination MAC address, data).

$$E_i \xrightarrow{(+\mathrm{MAC}_j,+\mathrm{data})}\!\!\blacksquare\ E_j \tag{5}$$

Messages are sent 'fire and forget' as impositions on to a remote receiver. While all interfaces generally promise to accept any MAC address, (unless they block with MAC access control) only the interface whose MAC address matches the destination in the message doublet actually promises to accept the message voluntarily. Note, however, that there is nothing other than convention to prevent all agents from accepting the data too; this 'promiscuous mode' is used for network intrusion detection, for example.

$$E_* \xrightarrow{-\mathrm{MAC}_j} E_i \quad \forall i,j \tag{6}$$

$$E_i \xrightarrow{(-\mathrm{MAC}_j,-\mathrm{data})\ \mathrm{if}\ (i=j)} E_j \tag{7}$$

Since the channel is unprotected, agents effectively promise the data to all others in scope. Moreover, all agents promise to decode the address and the data, but many will discard the results. Only the agent whose address is the destination MAC address promises to accept the data.

Any agents that are in scope of the transmissions can try to keep these promises. While the set of promises itself scales perfectly well, the assumption that every agent has to be in scope of every transmission does not scale, since it requires messages to be flooded to every node (agent), in principle. The primary issues with Ethernet today are that there are no ways to selectively limit the size of these flooding or broadcast domains. This makes the 'everyone please take a look at this' approach inefficient.

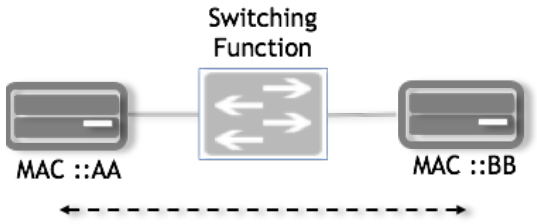

Fig. 1: An Ethernet switching function.

In figure 1 we see two interfaces that promise MAC address 00:00:11:11:11:AA (shortened to AA) and 00:00:11:11:11:BB (shortened to BB). Suppose we wish to send data from AA to BB, then, since the Ethernet is a push-based imposition protocol, only half a contract is needed for emergent delivery, and we leave the rest to trust[2].

$$\begin{array}{rcll} E_{\mathrm{AA}} & \xrightarrow{+\mathrm{MAC_{BB}}} & E_{\mathrm{switch}} & \\ E_{\mathrm{switch}} & \xrightarrow{-\mathrm{MAC_i}} & E_{\mathrm{i}} & \forall i \\ E_{\mathrm{switch}} & \xrightarrow{+\mathrm{forward\ MAC\ _{BB}}} & E_{\mathrm{BB}} & \end{array} \tag{8}$$

In each point-to-point interaction, the agent has to formally promise to use (-) the delivery service promised by the agent giving (+). This is the algebra of binding. There is no notion of a permanent virtual circuit.

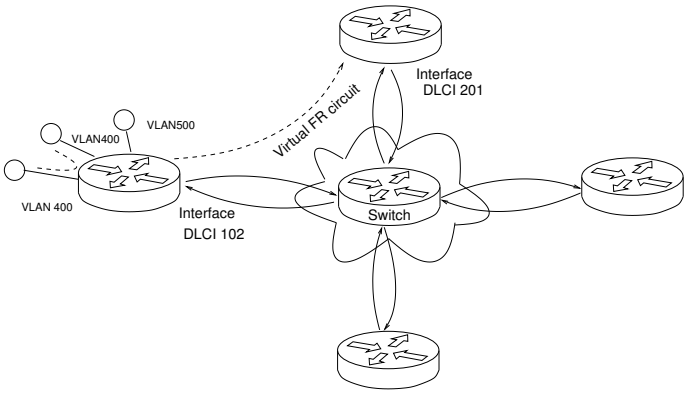

Fig. 2: Local and global virtual circuits, like VLAN, frame relay, ATM, MPLS, IP prefix are all based on container labelling.

We can compare the Ethernet approach to a reliable technology for a virtual circuit [BB14] (see figure 2).

B. Frame relay, ATM, etc

Virtual circuits are not limited to local areas and unreliable connections. Frame Relay was an early example of a reliable WAN circuit. It requires completely manual set up and tear down of routing or packet forwarding (this corresponds to keeping only part (2) of the automated forwarding promises in section III-C). On the other hand, it promises reliable handshaking rather than the unreliable 'fire and forget' approach of Ethernet. Interfaces in Frame Relay promised a unique name/identifier called a Data Link Connection Identifier or DLCI. Each interface is connected by a wire to a counterpart interface on a switch. The switch promises to forward traffic from one DLCI to another (see figure 2). Unlike the Ethernet case, handshaking is promised, as this is a *reliable* transmission.

$$\begin{array}{rcl} F_{102} & \xrightarrow{+\mathrm{DLCI\ 102\ if\ forward}} & \mathrm{F_{201}} \\ F_{102} & \xrightarrow{\mathrm{Use(forward\ 102\to 201),+DLCI102}} & \mathrm{Switch} \\ \mathrm{Switch} & \xrightarrow{+\mathrm{forward\ 102\to 201}} & F_{102} \\ \mathrm{Switch} & \xrightarrow{+\mathrm{forward\ DLCI\ 201\ if\ DLCI\ 102}} & F_{201} \\ F_{201} & \xrightarrow{\mathrm{Use(DLCI\ 102\ if\ forward)}} & F_{102} \\ F_{201} & \xrightarrow{\mathrm{Use(forward\ DLCI\ 201\ if\ DLCI\ 102)}} & \mathrm{Switch} \end{array}$$

In each point-to-point interaction, the agent has to formally promise to *use* (-) the delivery service promised by the agent *giving* (+). This is the algebra of binding.

Again, only half the circuit hand-shake is shown here, with uni-directional forwarding from the interface that promises DLCI 102 to the interface that promises to receive as DLCI 201. The remainder is fully symmetrical in reverse. The ATM and MPLS protocols have developed this idea further.

C. Internet (L3)

The Internet protocol dealt with the notion of Wide Area Networking by issuing two part addressing to cope with transmission scalability. IP addresses still promise to be unique in total, but are interpreted as doublets.

(network prefix, local address)

Only addresses with the same prefix are be considered in mutual scope for broadcasting, and messages addressed from one prefix to another have to be forwarded, with the help of deliberate promises to hand over a message, rather than 'flooding'. IP is thus a cooperative effort that builds on promises rather than impositions alone.

To make this work, IP needs two kinds of agent, which fall into different promise roles (see figure 3): *interfaces* (terminating connections), which only transmit and receive data intended for them, and *forwarders* (called routers or switches) that cooperate with multiple interfaces, and promise to selectively forward data from one interface to another out of a protected broadcast domain. This acts as a flood-barrier or firewall to packets promised to different prefixed networks.

To model routers, without giving up the interface abstraction, we introduce the concept of a route service (or link service), whose job it is to establish cooperative forwarding between the interfaces. This is the kernel.

Consider the example in figure 3. The source node has an address, normally written 128.39.78.4/24. As a doublet, the promises see it in two parts as $i =$ (prefix=128.39.78, local=4). We'll call this the source prefix, or, $j =$ (prefix=128.39.78, local=1) for the router interface. When a message is sent to an address with a different destination prefix, data are sent by imposition to the interface on the router with the source network prefix (usually the 'default route'):

$$I_{source_i} \xrightarrow{+(\mathrm{destination,local}),+\mathrm{data}} I_{\mathrm{router_j}} \tag{9}$$

Each router interface j promises the connected source interfaces i to use all such packets, a priori, and to present them

[2]This is what we understand as an *unreliable* service. Duplication of physical cabling (full-duplex) however makes it practically deterministic.

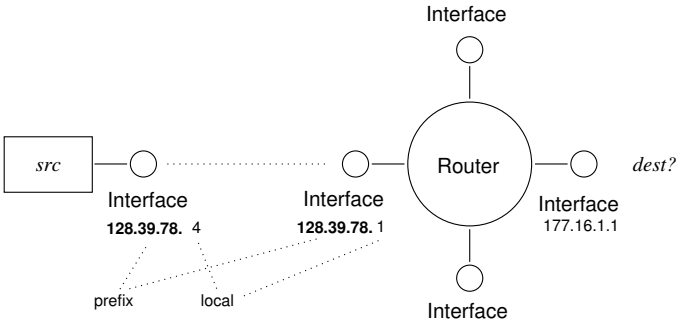

Fig. 3: Internet promises. An end-node or leaf and its single interface promises to relay through a 'router' which is surrounded by multiple interfaces, thus connecting multiple network branches.

to the router (kernel) which keeps the following promises.

$$I_{\mathrm{router_j}} \xrightarrow{-(*,*),-\mathrm{data}} I_{\mathrm{source}_i} \tag{10}$$

$$I_{\mathrm{router_j}} \xrightarrow{+\mathrm{prefix},+\mathrm{data}} \mathrm{Router} \tag{11}$$

Similarly, the interfaces connected to the router's interfaces promise to accept messages from the router that have their prefix:

$$I_{\mathrm{source_i}} \xrightarrow{-(\mathrm{prefix},\mathrm{source}),+\mathrm{data}} \mathrm{Router_j} \tag{12}$$

Crucially for messages to escape from a local region, the router promises all IP interfaces to forward messages it receives on one if its own interfaces according to a set of promises which we denote 'forward'. The router interfaces, in turn, bind to this promise by accepting it.

$$\mathrm{Router} \xrightarrow{+\mathrm{forward}} I_{\mathrm{router_j}} \tag{13}$$

$$I_{\mathrm{router_j}} \xrightarrow{-\mathrm{forward}} \mathrm{Router} \tag{14}$$

The forward promise has the following logic:

> (1) If the prefix of the destination interface is the same as the prefix of one of the router's interfaces, forward the message onto that interface.

The remainder of the promise requires configuration with knowledge of the wider world.

> (2) If the prefix of the destination interface is known to an internal database of external knowledge, i.e. the Routing Information Base (RIB)[3] forward the message to the interface known to lead to the desired destination.
> (3) Send all other message destinations to a predecided default interface, where we expect to reach some other router with greater knowledge of how to find the prefixed network.

Note that, like the Ethernet, this algorithm has only emergent behaviour that matches its design goal. It cannot, by direct imposition, assure a successful delivery of messages, because that requires the cooperation of potentially many intermediate interfaces and routing agents. In spite of this apparent lack of control, the Internet works demonstrably well. Trust plays a major role in operations.

D. VLAN: L2 channel containment

The concept of doublet addressing in IP enabled improved scalability, by black-boxing local networks, but added the cost of routing. How expensive routing is, in relative terms, is a constantly changing overhead that depends on many current technological factors. Fear of this cost tends to make datacentre traffic favour L2 solutions.

Routers were optimized for WAN delivery, so the obvious question for LAN managers was: could routing be simplified for smaller local regions without the paraphernalia needed for global routing? Further, could it all be done without the expense of managing IP domains, say at Layer 2?[4]

When packets don't have to be routed through multiple hops, i.e. when parts (2) and (3) of the forwarding promise can be ignored, a simpler form of prefixing can be used. This is the concept of the VLAN overlay. Interfaces can simply be classified, or tagged with short integer labels:

$$(\text{prefix, local address}) \rightarrow (\text{VLAN-id, MAC-address})$$

Then we have multiplet address components again (but now with a short VLAN tag instead of a large integer prefix), for boxing off local regions. A VLAN tag signifies membership in a private logical container. As with Frame Relay, these tags have to be configured manually, so routing is a human-centric process.

The concept of a Level 2 overlay has become quite popular for its perceived simplicity in small isolated networks. It's limitations have to do with scaling of the manual configuration and broadcast domains. VLAN is a brute force routing mechanism that scales linearly with the number of addresses in a container. Containers are not localized in physical space, only in logical channel space (unlike the assumed distribution of prefixes in IP). Thus this does not address the issues of physical scaling. However, we need something that scales like $\log N$ or better (like IP). We return to this in section VI-C.

E. Tunnelling addresses and transducer pattern

Embedding protocols inside one another is not the only approach to containment. One can also strip off and repackage data on different legs of a journey. To do this, one makes a transducer that converts one kind of addressing into another (see [BB14]).

ARP is one such service that maps between Ethernet MAC addresses and IP addresses. Instead of a physical forwarding table, a logical rewriting table is maintained. When a direct ARP conversion is not possible, data are sent to the default route, which is the address of the router interface I_{prefix} to the default interface. DNS is another transducer, that maps from symbolic addresses to IP addresses.

[3]The RIB (Routing Information Base) is kept in DRAM, the FIB (Forwarding Information Base) is usually a hardware lookup table on line-cards or in the ASIC and can be a distributed forwarding table; although the Router does tell FIBs when it updates a forwarding entry.

[4]The historical attempts to scale the L2 namespace have resulted in increasingly complex, expensive and brittle architectures with dozens of standard and proprietary protocols such as STP, RSTP, PVST, PVST+, MSTP, MLAG, VARP, VPC, Flexlink, LACP, VRRP, HSRP just to name a few examples.

The same principle has been applied to isolated networks, such as the reserved namespaces 10.0.0.0 and the example.com addresses 192.1.168.0/24. IP Network Address Translation (aka NAT) is now been promoted from crude workaround to viable technology, extending the local IP addressing component with additional internal addressing numbers, the rewriting outgoing addresses to point to the standard IP address range. End to end addressability is not normally promised in this scheme however, so it has limited value, (however see TRIAD [CG00]) for a viable scheme for extending IPv4 in this manner.

More recently, a tunnelling approach is also being used to artificially extend Layer 2 VLAN as a stop-gap measure for a technology users who are familiar with VLAN. VxLAN, and NVGRE are encapsulations of Ethernet L2 Frames, with tunnelling over IP to enable the physical reach across multiple gateways. Addresses add a multiplet component: a Tenant Network Identifier (TNI) or Virtual Tunnel End Point (VTEP) identifier embedded parallel channels.

These schemes perform two functions: i) they increase the number of possible VLAN-like channel addresses, patching a limitation in the VLAN implementation, and ii) they allow teleportation of broadcast domains across an IP scale network, transparently of routing concerns. Thus they do not eliminate the cost of IP routing, but offer a comfortable user interface for local network administrators.

IV. Promise principles for scalable networking

Let us consider the principles at work in the previous sections. We see two main issues: container-specific addressing, and interface to interface forwarding. Multiplet addresses allow the containment of broadcasts as well as selective routing of traffic by 'prefix'.

A. Addressability with scope or namespaces

By introducing multiplet addressing, we draw a logical (and perhaps physical) container around a network region which hides its internals with some kind of identifier or prefix, which acts as a namespace identifier. Everything inside the namespace is local and protected. There are two principles that explain these cases.

Principle 1 (Container multiplet addressing): Any system that promises to support n-tuple addressability of parts, for $n > 1$, enables logical or physical containment of information, as well as log-scalable routability between the containers. $\triangle$

To transmit data across multiple (possibly embedded) containers, we typically need an address component for each logical container. Thus interfaces a_i must promise to recognize one of the components a_i and pass on all others as passenger data:

$$\text{Interface}_\text{i} \xrightarrow{\pm(a_1,a_2,\mathbf{a_i},\ldots,a_n),\pm\text{data}} \text{Router} \qquad (15)$$

e.g. the a_i address components might include MAC address, IP address, VLAN number and VxLAN IDs. This set of addresses need to be configured and managed, either manually or by some mapping service. Some of these addresses overlap (like IP-LAN and MAC addresses).

Principle 2 (Forwarding by multiplet address): Forwarding of multiplet addressed data requires an infrastructure of forwarding promises by each members of each container for each address component in which all other components are ignored by other containers as payload data. $\triangle$

An interfaces a_i in a given container of level i would promise to accept other components addresses components as data only to be forwarded, not interpreted, i.e. as payload with no assumed semantics. In practice some of the address components might be removed or even rewritten, depending on the encoding as data traverse container boundaries, but that is not a requirement of the principle. All of the components have a continuing logical existence. It would be enough to ignore them. Note also that intrusion detection/prevention systems sometimes break the semantics of ignoring payload.

B. Addressable scalability

The scaling of multiplet addressing is a straightforward idea. The idea is to prevent a local namespace from becoming too large for flooding or broadcasting. The size of the namespace is limited either by a fixed number of nodes accessible in one multiplet address (e.g. VLAN tag, OSPF areas, BGP AS, etc), or equivalently, by the size of a prefix in a binary encoding of the multiplet (as in IP). In the first case, there is no defined limit to how many MAC addresses can occupy the same VLAN. Scaling is throttled by physical limitations. In the latter case there is an explicit quota tradeoff between local and global from a fixed number of addresses.

If an n bit address has a prefix of length p, this improves scaling through black-boxification of local regions. It transforms the addressing of $N = 2^n$ things into the addressing of merely $N_C = 2^p$, things globally and $n_C = 2^{(n-p)}$ things inside each of the C containers. That is $\log_{n_C}$ rather than n_C scaling.

There is no particular reason why IP addressing has to be limited to prefix quotas. IP Network Address Translation is an attempt to extend the range of local addressing, independently of the prefix quota space to alleviate IPv4 address depletion.

Multiplet addressing is a coarse graining of the network address space, through attendant abstraction. It transforms linear scaling into log scaling, or 'micro-management' (linear scaling) into group shepherding. Abstraction and trust provide logarithmic scaling over a scope of the log-base.

C. Addressable multi-tenancy

As a side-effect of supporting logical or physical containment one obtains the ability to support *multi-tenancy*. This can also be considered a mode of scaling in which one assigns containers to distinct organizational owners. This requires a mapping service too (like an ARP table for organizations). Registration of tenants is a manual human process. Presently there is IANA as a global directory service, ISPs for address delegation, and the Internet Exchanges for registering tenancy promises.

D. Two distinct network services

Networking supports two main use-cases:

- *Content delivery* or pull requesting (asynchronous retrieval promises of the form Node $\xrightarrow{X}$ Node).
- *Signalling* or push notification (synchronous impositions of the form Node $\overset{X}{\longrightarrow}\!\!\blacksquare$ Node).

The former is potentially a many-to-one association, for which we can employ versioning, replication (data-model denormalization), re-direction, and delocalization to good effect (e.g. Content Delivery Networks). Point to point addressing is less important here, and caching is highly meaningful. The concept of Name Based Routing has even been proposed to abstract away point addresses [BCA+12].

For the signalling, we still need point-resolution addressability, as signals cannot be cached, though they might need to be flooded. Service delivery generally involved a mixture of these two cases, which depends on the nature of the application being supported. Applications typically want to make certain promises about connectivity, security, e.g. load balancing and firewall filtering options

Application-oriented delivery suggests other forms of containment based on the logic of the service interaction. Current networking management abstractions make application specific requirements painful to configure because of lack of a consistent model for abstracting them. This brings us to the present day.

V. Generalizing network containers

Given that the principle of containment is so flexible and important for scoping data communications, it is natural to ask whether there are other more relevant abstractions that better support the needs of users. Several authors have opted to rethink network architecture [BHBS09], [BCA+12]. We consider this from a promise theory perspective.

A. Software Defined Networking

Software defined networking (SDN) is an umbrella term for a programmatic approach to managing network devices, using software controls to replace manual configuration [RCK+12]. It is considered more wide-ranging than the NETCONF or SNMP device management protocols of the past. SDN includes concepts related to network virtualization through overlays, network virtualization, virtualized L4/7 service insertion and control, management and orchestration of physical and virtual networking devices and functions (such as switches, routers), as well as control systems relying on specific control protocols like OpenFlow and OVSDB[5].

SDN is an interface transducer that essentially virtualizes existing concepts such as virtual end-points (computers), virtual cables (L1), virtual switches (L2), and so on. The initial motivating factors of SDN were to overcome the brittleness and the lack of programmability, manageability and agility in most network systems.

[5]Open vSwitch Database Management Protocol (OVSDB) is an OpenFlow configuration protocol designed to manage Open vSwitch implementations.

The inability to treat network infrastructure uniformly gave birth to the network virtualization through overlays, where underlying physical underlays are abstracted away in a way that provides perceived uniformity to the application/tenant traffic.

In some instances, SDN systems, like OpenDaylight Controller, have attempted to achieve what NMS (Network Management Systems) have struggled to do for last 20 years: provide a normalized access to network programmability regardless of underlying southbound control protocols. However, this virtualizes only the traditional concepts: virtual cables, virtual switches, virtual routers, etc.

In many SDN implementations, the control plane was fully removed from the networking devices and placed into a logically centralized controller – all without change of conceptual model.

With abstractions rooted in past technology choices, designed for the pleasure of network engineers rather than application designers, a cultural gap has been exposed between the needs of application programmers, writing increasingly distributed software. The current networking model requires decomposition into primitives like L2 networks (VLAN, VXLAN), tenant L3 domains (VRF) and multiple rules expressed as Access Control Lists (ACLs), or, in case of pure OpenFlow-based SDN, a set of flow rules.

Application architects think about application components and component interactivity, they rarely think about networks, firewalls and other L4-7 services of their own free will. Instead, they think about services providing *functions* to other services, while consuming functions of the infrastructure and other application components/services.

With the emphasis on programmability of existing technology kludges, the business purpose of an application can quickly be lost in low level details. This makes application design and updates a painful process.

Numerous proposals have attempted to solve this problem, with most approaches addressing taking issue with process inflexibility and some reduction to the richness of network functionality. Current SDN solutions seem unconvincing in addressing these issues, based on legacy concepts.

B. Application-Oriented Network Promises

Another approach is to rethink the way architects reason about interaction with the network altogether. Application-centric group-based policies could be used for consistent enforcement of service requirements. Such 'requirements' (impositions) can also be turned around as promises, specified by the application architect. This makes a concise self-documentation of purpose, so it also has positive semantic value.

Promise Theory provides a theoretical framework for such policy abstractions too, and so can be instrumental in solving a larger problem: how to build a scalable, self-stabilizing network, supporting any kind of abstraction.

There is no particular reason why the container principle cannot be applied to other logical elements than channels and network boundaries. A piece of software can also be viewed as a local network that connects to a wider area.

Application architects need to think of network behaviors as application components (the application services) and manage their interactivity. Application architects can then focus on understanding what functions a given service provides and relies on through exposed interfaces.

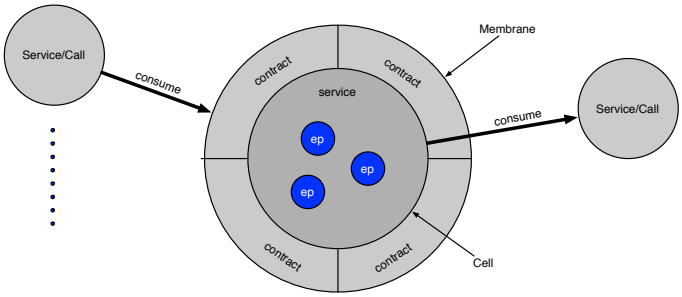

Fig. 4: Layers in an application oriented model. This based on the Insieme

Such an application service might be thought of as a cell or logical network container, encapsulating a number of application servers that provide the service contained within. All server end-points within a service promise a set of functions to the consumers of the service. The collection of such (+) promises can be thought of as the basis for a contract by which others interact with this service. In a cell analogy, this could be called the cell membrane. The cell membrane protects and regulates the conversations in and out of the cell. The (-) promises that listen for client impositions could be thought of as 'receptors' that identify what services can be talked about and how.

Consider a classical three-tier application made from cells (promise agents clusters):

- WEB - Web services.
- APP - Application services (e.g. Tomcat or jboss)
- DB - database services.

This application tier breakdown is another containment pattern (like L2, L3 etc) that might eventually be superceded by another, due to changes in technology; but there will always be some schematic level at which one can sketch a functional system. The super-agent clusters now become the service provider tiers. These form ±promise bindings (i.e. cellular contracts) to make an end-to-end flow of logic.

$$T_{\mathrm{DB}} \overset{\pm\mathrm{db}}{\longleftrightarrow} T_{\mathrm{App}} \overset{\pm\mathrm{app}}{\longleftrightarrow} T_{\mathrm{Web}} \overset{\pm\mathrm{web}}{\longleftrightarrow} T_{\mathrm{User}} \tag{16}$$

Addressing, forwarding and even transducing data from one tier to another is not fundamentally different from repackaging Ethernet frames into IP datagrams. There are transducers for these functions that are designed to be fit for purpose in the context of the application. It is this adaptation to context that we do not see in traditional networking.

The tiers have internal structure, but insofar as we only interact with their promises, we don't care what it is. Each tier comprises multiple computational end-points or service 'hosts'. A host or end-point might be a Virtual Machine (VMs), a container (like LXC or Solaris Zones) or simply a bare-metal compute instance with a network connection. Since all hosts within a tier provide identical function within the application, they form a 'promise role' group that makes identical security, forwarding and QoS promises. Membership of these end-points within a group/role can be either administratively or automatically established, most commonly based on the properties of the compute element (like VM attributes).

The infrastructure promises made between these tier-agents have to be established. To avoid the kind of manual setup problems of FR, VLAN, CLI, etc, the end-user's needs are communicated by some kind of signalling impositions (API calls), e.g. 'I need the db to talk to the app and the firewall needs to be opened, I want sufficient channel capacity ("bandwidth") to be promised'. As long as the service accepts such impositions from clients, it would try to promise what was required. Only the service has the necessary knowledge about whether such a promise is plausible however, so the decision belongs inside as an autonomous decision.

Each tier is really a collaborative group of agents that not only promise to provide and user service between one another, but which internally promise to collaborate uniformly to keep coordinated promises (see figure 8):

$$\begin{aligned} \{A_{\mathrm{DB}}\} &\overset{\pm\mathrm{db}}{\longleftrightarrow} \{A_{\mathrm{App}}\} \\ \{A_{\mathrm{DB}}\} &\overset{C(\mathrm{DB})}{\longleftrightarrow} \{A_{\mathrm{DB}}\} \\ \{A_{\mathrm{App}}\} &\overset{C(\mathrm{App})}{\longleftrightarrow} \{A_{\mathrm{App}}\} \end{aligned}$$

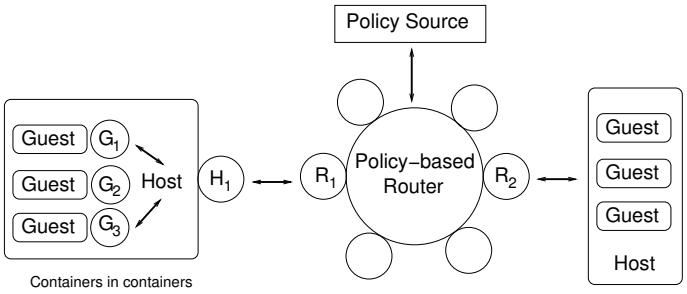

Fig. 5: A more generic example of the principles of containment and forwarding, with policy specific promises, leads to a pattern like this. Host containers promise to forward communications between guests and 'outside' as 'routers'. Two levels of container means agents need up to maintain triplet addresses (Guest, Host, Outside). A single policy source acts like a calibration source. All controllers are embedded in the containers.

The challenge of this abstraction is: how do we create a unified user experience around this for application engineers? Networking promises have traditionally been built around the technology containers of the OSI model. However, policy-based configuration can be defined within logical containers, based on pattern matching of local attributes [Bur95].

Users need to comprehend a design composed of an increasing number of logical entities and layers, arising from cross-cutting abstractions. We believe the answer here lies in the managing patterns of promises, which can also be described in terms of the container principle. The challenge for technologists is to avoid the temptation to push complexity

back onto users. If one can avoid referring to the OSI layer technologies altogether and move towards a description based on Service Level Agreements (SLA). Current SDN makes this accessible but not natural. The key would seem to be some kind of promise compiler.

C. End-to-end service promises via proxy

To see how this could be done, we return to promise theory. The 'proxy' or intermediate agent pattern was described in [BB14] abstracts the end-to-end delivery promise of a service S through some promise to handle the delivery details by proxy P to a client. Both client and server may be guests running inside various containers (see figure 6) .

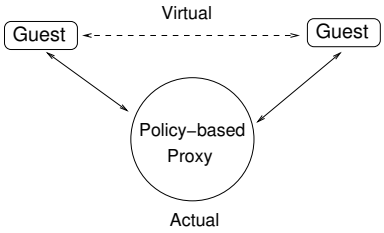

Fig. 6: From a user perspective, applications just want to talk to applications without knowing about the stack of containers. Container issues can be handled by modelling a generic proxy that promises to mediate this connection, with no internal details exposed..

The abstraction we would like to expose to the user is for logical services and consumers to simply make promises directly to one another, without worrying about all the intermediate agents in between. The proxy pattern shows how this can be achieved through the following promise pattern, repeated for each logical service

$$\begin{aligned}
\text{Server} &\xrightarrow{+S(P)} \text{Client}\\
\text{Server} &\xrightarrow{-P,+S} \text{Proxy}\\
\text{Proxy} &\xrightarrow{+P,-S} \text{Server}\\
\text{Proxy} &\xrightarrow{+P(S)} \text{Client}\\
\text{Client} &\xrightarrow{-S(P)} \text{Server}\\
\text{Client} &\xrightarrow{-P(S)} \text{Proxy}
\end{aligned}$$

We refer readers to [BB14] (section 11.3) for a discussion of these six promises.

Examples of the service S could be: to provide connectivity over a secure channel, to grant or deny access to data, to commit to or retrieve from storage, to provide web transport. It is important to note that a go-between might create a superficial similarity of function, but it also adds four promises and hence four possible points of failure to the equation.

In terms of the cell membrane analogy described above, we would say that the outer-membrane promises identify how others can communicate to that service, what they can communicate about, and what happens to the traffic when they communicate. Today, one would have to separate it into VLANS, into Firewall rules, Load-Balancing rules etc. These are details we would prefer to leave to a proxy on imposing on it a minimum of application specific requirements.

By providing this as a service, one is able to reason about the application, and how it interact with other applications rather than the physical underlay. The membrane thus provides a level of abstraction that hides the details of the cell composition[6].

By isolating promises as containers that promise to play certain roles in an application design, one can think about the Datacentre as an organism. Then an organism comprises of many boxes containing multiple functions that manage their own resources based on a policy declaration (see figure 7).

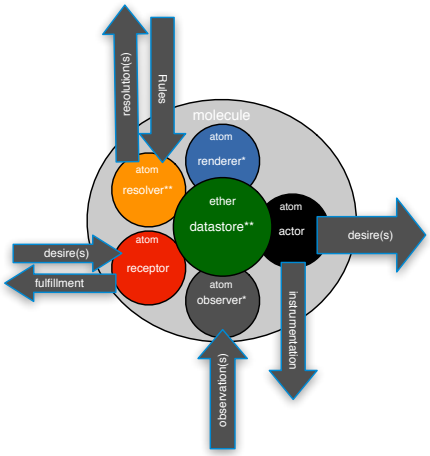

Fig. 7: Applying the encapsulation principle to all internal services, each with their own promises, allows us to abstract away implementation details, exposing only relevant promises.

D. Policy-based containment

Putting the pieces together, what one ends up with from a promise perspective is a set of logical containers that keep user-friendly promises and conceal engineering details. This is not an imposition control system, but rather a scaled design that compiles into continuously enforcable local low-level configuration promises. Based on net-wide coordinated policy, one compiles high level promises into a kind of assembler code for the low level network configurations supporting a business purpose.

Cells promise to be smart enough to engage with each other if they need to coordinate their activities. But they do not need to understand detailed semantics up and down the container layers. Nor does there have to be continuous access to centralized services or data. The common theme is to scale policy in an architecture-independent way.

VI. Fitness for purpose

We have shown how to represent both low and high level intentions in terms of promises made by autonomous entities (agents) in IT networks. We have also indicated how a distributed application can itself be viewed as a service-oriented network built from autonomous components. The container principle serves both to abstract and limit the effect

[6]This is how Insieme's architecture works, which is currently under submission to OpenDayLight.

of irrelevant load on inappropriate parts of such collectives. Containers, not merely as abstract layers, but as namespace boundaries for autonomous agents to work in, are clearly a key abstraction for scalable service-oriented data communications.

The next obvious question to ask is: is a set of network containers *fit for purpose*? How indeed might we align an abstracted networked system with a given business purpose?

A container (a promise super-agent) can be identified by the pattern of promises that are made to other agents beyond its 'border' or 'membrane'. This promise pattern becomes associated with the *role* of the container [BB14], just as any namespace label can act as a role identifier. Containers that play similar roles make functionally equivalent promises.

Promise Theory answers the question of alignment simply in terms of these outward intentions. Suppose we can compile our desires for an application or goal into a number of promises with bodies $d_1, d_2, d_3 \ldots$ that would need to be kept. Then, if our network actually makes promises:

$$\text{ApplicationNetwork} \xrightarrow{b_1, b_2, b_3 \ldots} \text{Users} \tag{17}$$

Then we can say, straightforwardly, that the network is aligned with the purpose if

$$d_1 \cup d_2 \cup d_3 \ldots = b_1 \cup b_2 \cup b_3 \ldots \tag{18}$$

The outward promises form the user interface to the containers. Clearly application oriented containers are more directly business-purpose friendly than OSI-layer containers for L2 or L3 abstractions. But there is more to business purpose than just functionality. Other systemic promises need to be considered too when discussing fitness for purpose. Will the network architecture scale? Will it fail gracefully under pressure? Will it respond to our needs? Many of these answers follow from the graph-theoretic properties of the promise graph. Let's consider these in briefly turn.

A. Distributed control

Control is the ability to exert influence over the parts of a system. Traditionally, one looks for:

- A single point of remote control, or
- A single point of policy calibration.

In promise theory, control can only come from within an agent as an autonomous decision to promise something; for instance, a promise to provide a service $+S$, such as a function or behaviour, or a promise to accept a service $-S$, like an Access Control List promise.

Since an autonomous agent is free to ignore signals and impositions from outside, remote control requires an explicit promise by all agents to accept impositions from a single point of command. This is fragile and it is a serial signalling regime. A more efficient approach is to turn the signalling problem into a parallel content delivery problem by distributing predecided policy that the distributed agents can accept and cache for apply self-control. Thus, Promise Theory reveals a policy model to be an improvement in reliability, fault tolerance and scalability.

With a central controller approach, the central entity has to make regular impositions on different parts of the system. This adds complexity from outside the system and makes the controller point location a bottleneck. A promise based approach allows self-control based on a cached policy, without further signalling. Thus it avoids to cost and resilience issues of signalling infrastructure. Since policy is a set of pre-decided promises with associated context which has potentially global scope, it does not have to be communicated in real time, it can be cached and replicated efficiently.

B. Resilience and fitness for purpose

Redundancy plays two roles: one in scaling, for parallel throughput of data, and another in plasticity or failover resilience of agents [Bur13]. Traditionally one thinks of:

- Single points of failure (unique fragile point).
- Single point of contact (redundancy allowed).

Redundancy means extending service resources inside a service container in a fashion that is transparent to a client (see the pattern in figure 8).

In promise theory, the generative rule is that no agent may promise anything that is not about its own behaviour. Thus resilience also has to come 'from within'. Each agent promises its own responsibilities within the whole. For $+S$ promises each agent can only do its best and presume additional support from others it knows nothing about. For $-S$ promises to consume a service agents must say something like 'I promise to use the service S from either agent X or Y or Z' (not just X). The promise principle forces the responsibility for resilience back onto the end-point, i.e. client or server, to the edges of the network. No middle boxes like load balancers can improve the situation, because no agent can influence their behaviour. The redundancy pattern is shown in figure 8. The

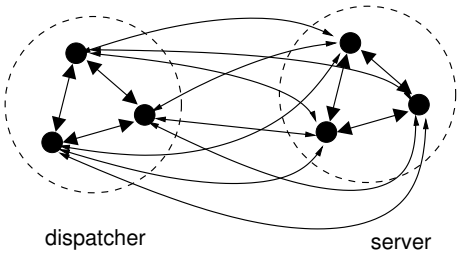

Fig. 8: Avoiding single points of failure, increases the amount of internal structure a remote agent needs to know about. A single point of contact for each tier breaks scaling and resilience, even with redirection.

fault tolerance can be understood as having clusters of hosts working cooperatively as 'super-agents' (see the dotted circles) to keep the same forwarding promise (see figure 8). Each agent in a circle promises to cooperate with every other with mutual cooperation promises Each agent promises to try to failover to other agents; thus load balancing also becomes a client function (without intermediaries). These promises are typically roles by association [BB14] (e.g. by policy coordination) rather than physical data links. The Clos networks (see fig. 9) are realizations of these, in varying degrees of approximation.

This pattern can be applied both physically and logically between any pair of roles: client/server, interface/switch, network/router, etc. For a network, the intra-agent promises inside

each super-agent do not have to be physical connections, but rather intentions to act symmetrically. Note however, the architecture reported at Facebook to formalize this [fac]. If one has multiple servers or end-points they should keep the same promises so that users cannot distinguish between them.

A more normal limited rendition of this pattern is found in the order n Clos architecture (see fig. 9), which satisfies this principle in the following way. Since only point to point connections are used (and there are no intermediaries) the switches themselves are the agents, and the only option for resilience is for each switch to take responsibility and promise and use ($\pm$) service from multiple consumers and providers, leading to the overall redundancy [AFLV08].

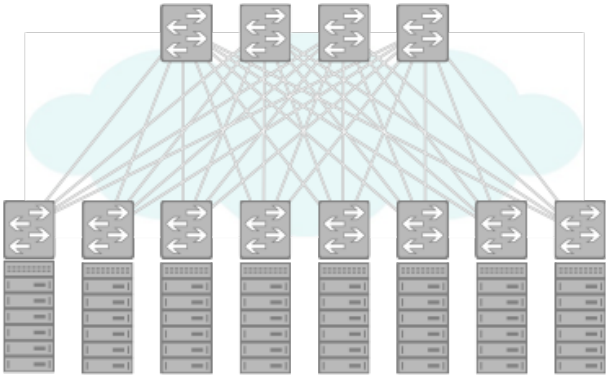

Fig. 9: A Clos network is a logical embodiment of figure 8. Note: the cloud depicts zero or more additional layers of CLOS switches.

If the goal of containment is to reduce complexity, then this architecture may be promised by the simplest of patterns:

$$\{L_{c,l}\} \xrightarrow{\pm\text{data}} \{C_{c,l}\}, \quad \forall l, c \tag{19}$$

$$\{C_{c,N_c+l}\} \xrightarrow{\pm\text{data}} \{S_{s,c}\} \quad \forall c, s, \tag{20}$$

If the promises made for communication are this simple, then the complexity of applications can also be eliminated without abstraction, by employing simple stock patterns, like this or like a lattice. For example using a simple configuration engine to enforce the distributed pattern. Complex management interfaces with extensive programmability become unnecessary.

C. Scalability of networks

Scalability of networks is a complex issue that describes how a system can support increased usage, component addressing, data throughput, and all without rapidly growing latency.

- Physical connectivity need sufficient serial/parallel channel throughput capacity.
- Low latency and forwarding cost at routers, switches and protocol transducers.
- Sufficient plasticity, i.e. if something breaks there is a failover safety net.
- Human comprehensibility is a major limitation favouring centralization and .

These are all systemic promises of an architecture. They are emergent features of a network, and cannot be controlled from any single point.

The Clos or Fat Tree architecture is also used today for horizontal scaling of traffic inside datacentres. The advantage of a mesh formed from small autonomous parts is that it can be extended indefinitely and repaired easily with only a small failure domain. However, promise analysis does not suggest that it is a panacea in spite of its popularity. A Clos network is basically a redundant array of stress concentrations, where load is over-subscribed on the assumption that chance will favour the gamble. It is an expensive architecture.

Both Clos and lattice physical architectures suffer from the intermediate agent problem, namely that they delivery by proxy. The number of points of failure increases with distance, and the cost of redundant coverage increases exponentially. For a lattice, complexity is lower but the number of hop cost might be larger. A proper examination of these economics is probably overdue.

VII. Conclusions

Promise Theory is about describing service-oriented functionality with the avoidance external programmatic reasoning in networks of autonomous agents. Complexity and fragility may also be avoided through a simple desired end-state model of intent modelled on promises. We have attempted to use the Promise Theory to make neutral assessments of existing networking principles, to lay bare the functional and emergent aspects. We hope that, by summarizing networking in this manner, the challenges and solutions appear plainer, and not insurmountable.

Some of the problem we've pin-pointed include:

- Problems in scaling the protocols and their management, especially in the datacentre NS-EW. Promises reveal that control is fundamentally a trust issue. Low trust micromanagement scales poorly.
- Problems of communicating through meaningful abstractions in the context of applications.
- Problem of labelling and isolating (or at least tracing) resources used by different application owners.

How one chooses to solve these issues is another matter. However, here too, the concept of promises suggests an approach based more on decentralized abstraction Bottom up design favours stability over functionality, but purpose drives change and this comes from the top down. Striking this balance is a task for new container abstractions and simplifying patterns.

Paul Baran foresaw an Internet that was a lattice-like mesh [Bar60]. Alas, research shows that network growth occurs by accretion into clustered power-law structures, which is a basically fragile pattern. Clos architectures have this fragility too. Lattices rather than Clos structures are what nature has chosen for scaling its own infrastructure, just as decentralized rather than centralized is the outcome of natural selection in most (but not all) cases. This bears careful re-examination so that mere habits do not dictate best practice going forward.

What we believe the network industry has done right in the past is to build network infrastructure on a desired end state model with a low level of reasoning. There is no programmatic reasoning needed to run OSPF or VLAN for instance, only a few fixed data-driven promises. What it has done wrong is

in being closed and inflexible about how to configure those promises and orchestrate entire architectures for the purpose of running applications, forcing business users to confront irrelevant details.

Complexity is a major part of the challenge facing the industry. This is a cognitive issue rather than a physical one. How shall we deal with that? By returning to "easy" ideas like centralization and brute force that scale poorly we might be able to stave off dealing with the issues, but at what cost to resilience and fault tolerance?

There are many topics we have not been able to cover in this paper. A full scalability and economic analysis of current datacentre patterns in relation to different kinds of application architectures and scales also seems overdue. Today we hear mainly about the massive social media datacentres. While this might be relevant for shared cloud services, there are still questions to be answered about how best to scale physical networks for different kinds of applications.

It would be interesting to compare and contrast the different approaches developed by ourselves in regard to these challenges, and see how they could be sewn into a unified view. How does IPv6 fare in the current picture, what about MPLS and other technologies? How shall we identify fitness for purpose without losing comprehensibility of networks?

Because of cloud patterns and the resource reusability they enable, it has become common to say that datacentre traffic patterns are changing from traditional "North-South" to "East-West" due to the nature of compute virtualization. However, these concepts might themselves be due for re-evaluation.

As many authors have commented, there is much to be learned about scaling from biological systems [Mar98], [Ger07] because biology has clearly evolved systems that embody the properties we find desirable for building robust networks. Although biology may appear complex, its underlying principles, such as degeneracy and exploratory behavior are extremely simple, and easy to model in our promise-theory based approach. With a promise model, and a network of autonomous systems, each agent is only concerned with assertions about its own policy; no external agent can tell it what to do, without its consent. This embodies biological scaling too.

IT services are no longer just the domain of software engineers who run on top of incidental hardware. The challenges of scaling data communications are amongst the most difficult we now face. We believe that these are crucial and exciting times especially for network innovation as all aspects of infrastructure become key players in the design of society's IT systems.

ACKNOWLEDGMENTS

MB and PB would like to thank Dinesh Dutt for illuminating conversations. MB would like to thank John Willis.

REFERENCES

[AFLV08] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM Computer Communication Review*, 38(4):63, October 2008.

[Bar60] Paul Baran. Reliable digital communications systems using unreliable network repeater nodes. Technical report, RAND Coorporation, 1960.

[BB14] Jan Bergstra and Mark Burgess. *Promise Theory*. χtAxis Press, February 2014.

[BCA+12] Md. Faizul Bari, Shihabur Rahman Chowdhury, Reaz Ahmed, Raouf Boutaba, and Bertrand Mathieu. A survey of naming and routing in information-centric networks. *IEEE Communications Magazine*, 50(12):44–53, 2012.

[BHBS09] Mahesh Balakrishnan, Joe Hoffert, Ken Birman, and Douglas Schmidt. Rethinking reliable transport for the datacenter. In *Proceedings of the Large-Scale Distributed Systems and Middleware Workshop (LADIS 2008)*, September 2009.

[Bur95] M. Burgess. A site configuration engine. *Computing systems (MIT Press: Cambridge MA)*, 8:309, 1995.

[Bur05] Mark Burgess. An approach to understanding policy based on autonomy and voluntary cooperation. In *IFIP/IEEE 16th international workshop on distributed systems operations and management (DSOM), in LNCS 3775*, pages 97–108, 2005.

[Bur13] Mark Burgess. *In Search of Certainty - The Science of Our Information Infrastructure*. χtAxis Press, November 2013.

[CG00] David R. Cheriton and Mark Gritter. Triad: A scalable deployable nat-based internet architecture. Technical report, 2000.

[fac] Facebook's Data Center Network Architecture. *IEEE Optical Interconnects Conference (OI)*.

[Ger07] Gerald Jay Sussman. Building robust systems - an essay. Technical Report An Essay, MIT, January 2007.

[LWH+12] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldmann. Logically centralized?: State distribution trade-offs in software defined networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 1–6, New York, NY, USA, 2012. ACM.

[Mar98] Mark Burgess. Computer immunology. In *Proceedings of the 12th Systems Administration Conference (LISA '98),*, Boston, December 1998. USENIX Association.

[MB04] G. Canright M. Burgess. Scalability of peer configuration management in logically ad hoc networks. *Network and Service Management, IEEE Transactions on*, (1):21 – 29, 2004.

[MK05] Mark Burgess and Kyrre Begnum. Voluntary cooperation in pervasive computing services. In *LIUSA'05*, 2005.

[RCK+12] Barath Raghavan, Martín Casado, Teemu Koponen, Sylvia Ratnasamy, Ali Ghodsi, and Scott Shenker. Software-defined internet architecture: Decoupling architecture from infrastructure. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 43–48, New York, NY, USA, 2012. ACM.

[Wil06] Frank Wilczek. *Fantastic Realities: 49 Mind Journeys And a Trip to Stockholm*. World Scientific Publishing Company, Hackensack, N.J, 1 edition edition, March 2006.