Decentralized Software Architecture

Rohit Khare
University of California, Irvine
rohit@ics.uci.edu

December 2002

ISR Technical Report # UCI-ISR-02-6

Decentralized Software Architecture

Rohit Khare
Institute for Software Research, University of California, Irvine
Irvine, CA 92697-3425
rohit@ics.uci.edu

ISR Technical Report # UCI-ISR-02-6

Abstract: A centralized (or even distributed) system admits only one correct answer to a question at a time. In contrast, a decentralized one allows several agents to hold different opinions, all equally valid. While the term ‘decentralization’ is familiar from political and economic contexts, it has been applied extensively, if indiscriminately, to describe recent trends in software architecture towards integrating services across organizational boundaries.

This technical report investigates how ‘decentralization’ can be defined in the context of software architecture; provide a formal model of two causes of decentralization: latency and agency; review historical trends forcing decentralization in software; as well as design, implement, and evaluate a proposed software architectural style called DECentralized Event Notification Transfer (DECENT).

We believe its principal contributions will include: formally exposing the often-tacit requirement for simultaneity between components; novel rationales for adopting event-based communication, standardized application protocols, and stateless messaging; describing application-layer internetworking of software services across several routers, (rather than a central bus); a microkernel-like refactoring of traditional messaging middleware on top of a basic event model; techniques for using consistent hashing to increase reliability, availability, and scalability of decentralized services; and evaluation of the effectiveness of DECENT-style architecture for enabling 3rd and 4th-parties to add properties such as security and interoperability without modifying the original services.

Decentralized Software Architecture

Rohit Khare
Institute for Software Research, University of California, Irvine
Irvine, CA 92697-3425
rohit@ics.uci.edu

ISR Technical Report # UCI-ISR-02-6

Abstract

A centralized (or even distributed) system admits only one correct answer to a question at a time. In contrast, a decentralized one allows several agents to hold different opinions, all equally valid. While the term 'decentralization' is familiar from political and economic contexts, it has been applied extensively, if indiscriminately, to describe recent trends in software architecture towards integrating services across organizational boundaries.

This technical report investigates how 'decentralization' can be defined in the context of software architecture; provide a formal model of two causes of decentralization: latency and agency; review historical trends forcing decentralization in software; as well as design, implement, and evaluate a proposed software architectural style called DECentralized Event Notification Transfer (DECENT).

We believe its principal contributions will include: formally exposing the often-tacit requirement for simultaneity between components; novel rationales for adopting event-based communication, standardized application protocols, and stateless messaging; describing application-layer internetworking of software services across several routers, (rather than a central bus); a microkernel-like refactoring of traditional messaging middleware on top of a basic event model; techniques for using consistent hashing to increase reliability, availability, and scalability of decentralized services; and evaluation of the effectiveness of DECENT-style architecture for enabling 3rd and 4th-parties to add properties such as security and interoperability without modifying the original services.

1. Introduction

Like any other design discipline, software development is subject to the vagaries of fashion. In recent years, there has been a surge in the popularity of the term 'decentralization': we hear of 'decentralized file-sharing,' 'decentralized supercomputers,' 'decentralized namespaces,' and a slew of similar claims of 'peer-to-peer,' 'Internet-scale,' and 'service-oriented' architectures [MKL+02].

Such interest is perhaps inevitable in the wake of the unquestionable success of World Wide Web for enabling 'decentralized hypertext' [BC92]. This suggests closer investigation of how the Web's software architecture, as described by REpresentational State Transfer (REST) [Fie00], actually supports or inhibits decentralization of software applications.

Naturally, we must begin with a clearer understanding of the term 'decentralization.' The following subsections provide a definition; examples of centralized, distributed, and decentralized social behavior; manifestations of decentralization in software artifacts; and an outline of the remainder of our topic proposal.

1.1. Defining 'Decentralization'

> ***Decentralized*** (adj): withdrawn from a center or place of concentration; especially having power or function dispersed from a central to local authorities.
>
> ***Decentralization*** (n): the spread of power away from the center to local branches or governments [WN02]

Perhaps the subtlest aspect of these definitions is hinted at by the verbs 'withdrawn' and 'spread': *one can only decentralize what was once centralized.* Critically, this introduces an observer's stance: deciding whether a given phenomenon is centralized, distributed, or decentralized is thus a matter of perspective.

Consider an online auction service. To a buyer, it appears to be a decentralized marketplace, since anyone can list goods for sale. To a seller, it appears to be a centralized auctioneer, since it alone determines the order and validity of bids. Moreover, to an engineer, it appears to be a distributed database service, since the data is stored and processed on many servers working in parallel.

Furthermore, it is important to distinguish between independent systems and decentralized ones. It is not sufficient to state that a set of restaurants, say, can set their own prices for a cup of coffee. In a capitalist system, if they are each owned by separate agents, they have complete freedom to set prices *without* reference to each other. There is no centralized price-of-coffee in the first place.

However, if those restaurants are part of a franchised chain, the power relationship is quite different. There *is* a concept of a centralized, franchise-wide price-of-coffee, even though the final authority to set actual prices has been "dispersed from a central to local authorities" as local conditions warrant.

1.2. Decentralization in society

'Decentralization' was a political and economic concept long before it was ever applied to describe software. Thus, the essential anthropocentrism of the term's traditional definition suggests exploring a social process that illustrates the phenomenon: markets.

There are many different kinds of markets, but they all provide a pricing mechanism to achieve equilibrium between supply and demand. For example, the New York Stock Exchange (NYSE), National Association of Securities Dealers Automated Quotation System (NASDAQ), and the over-the-counter bulletin boards ("Pink Sheets") are three different markets for trading shares of stock. Each employs a different pricing mechanism.

1.2.1. Centralized Markets: NYSE

A completely centralized solution would be for every trader to gather under a buttonwillow tree at the same time and tell one person the prices they were willing to trade at, and then let that person announce a single, equilibrium price. That tradition continues unbroken to the present day, at the very same location, in the form of the NYSE specialist.

Every single order to buy or sell on the NYSE from around the world eventually filters up to a single human being. That specialist, in turn, is within earshot and line-of-sight of a handful of member brokerage firms' representatives on the NYSE trading floor. Aided by little more than 19th-century telephones and tickers, this handful of people orchestrates the exchange of billions of shares every day.

NYSE proponents argue that a centralized system is the most efficient, since every trading order is 'exposed' to the widest range of counterparties. The tradeoff is that prices change slowly, since the pricing mechanism requires comparatively-slow human judgment.

1.2.2. Distributed Market: NASDAQ

The NYSE only lists a small fraction of publicly traded companies, however. Far more are listed on the virtual NASDAQ market, which has no physical trading floor. Instead, individual traders forward orders to a small number of 'market-makers,' broker-dealer firms that commit to using a shared computer system to track all of their outstanding orders.

Proponents of the NASDAQ market note that it is much cheaper to replace the role of a centralized specialist is replaced by a distributed consensus among several firms. The resulting efficiency enables far more firms to be listed. The tradeoff, however, is that determining the best price requires time for all the other market-makers to be consulted (currently limited to 30 seconds).

1.2.3. Decentralized Market: Over-the-Counter

In turn, there are even more stocks that are too cheap and too rarely traded for either of the other two pricing mechanisms. So-called 'penny stocks' do not have a single fair value across different brokers. In an over-the-counter market, each transaction is independent, so the same trade at the same time can clear at *different* prices. Instead, every few days, brokers publish their recent prices for these stocks, which were mailed out on 'pink sheets,' a term still in use today.

If this makes over-the-counter markets seem like a poor compromise for thinly-traded goods, consider that the largest marketplace in human history is also decentralized. Every day, a trillion dollars' worth of foreign currencies are exchanged, around the world, around the clock, and essentially without regulation.

While there are many tradeoffs to such a decentralized market – the biggest players, known as "money center banks" can manipulate markets in ways considered illegal in stock trading – it is nearly completely fault-tolerant, proving more scalable than centralized markets in theory and in practice.

1.3. Decentralization in software

The consequences of decentralization for software may be less familiar than for economics or politics. Beyond the obvious challenge of modeling and automating decentralized social processes – applications for trading stocks or currencies, say – there are also subtler signs of decentralization in the construction of software itself: component identification, communication protocols, storage formats, and authorization models, are only a few ways disagreement inhibits software integration between organizations.

Consider something as simple as a timestamp. Humans have many equivalent ways to format a date string, but if we expect several software components to interoperate, we need to shift from a model where the very format of a Date: header is subject to decentralized control of any counterparty. After all, you can't prevent someone else's application from generating dates marked "2002 A.D.," but you ought to expect your software to interoperate with it. This is the basis of the classic Postel dictum for Internet applications: "Be liberal in what you accept, and conservative in what you generate."

Nor is decentralization merely a design-time constraint; its impact may only be felt years after a system is deployed. Well into the '80s, Internet standards permitted two-digit dates. That did not prevent organizations from upgrading to Y2K-compliant implementations, without impairing interoperability with once-valid services.

From this perspective, the original software as shipped coped with all the known sorts of date formats, but decentralization ultimately permitted counterparties to invent new formats. The longer a system is expected to be in continuous operation, the more careful a software architect must be to predict and accommodate decentralized control of even 'obvious' agreements.

It could be said that the state of the software industry today is the realization of a sustained, common vision for "assembling software out of building blocks." Ultimately, though, this paradigm is limited reusing software within organizational boundaries: today's multibillion-dollar Enterprise Application Integration (EAI) market is limited to assembling ever-larger applications under the control of a single organization.

Instead, we believe the industry's common vision is moving on towards "assembling software out of services." Unlike components, which may be designed separately but are owned and operated by the same organization, services can be designed, owned, and operated by separate organizations. Furthermore, rather than identifying concrete "applications" that can be brought up, shut down, or upgraded under the control of a single agency, an "application network" is intended to operate continuously and requires dynamic, multilateral architectural evolution.

If the future of software requires integrating services running on computers that are very far away and owned by others, then decentralization – forced by high latency and diverse agency – will become a dominant concern for software architecture.

1.4. Summary

Decentralization poses a fundamental challenge to the design of software: variables could have several simultaneously valid values. Either a decentralized variable – say, the yen-dollar conversion rate – will have to be reduced to several centralized or distributed equivalents – the yen-dollar conversion rate *according to* Bank A, Bank B, and so on – or we must envision a new style of software architecture that explicitly acknowledges that a single resource can have multiple valid values.

This approach is seemingly at odds with our most fundamental abstractions of computing devices. Even a Turing machine assumes a tape is either marked or unmarked, and that it can't magically change between steps. Formally, we introduce a definition of simultaneous agreement, a test of whether two observers can compute using the same values at the same time. Since agreement is impossible on asynchronous networks, and still takes at least the maximum possible delay on any actual network, we argue that decentralization is inevitable once simultaneous agreement becomes infeasible.

There are two basic reasons simultaneous agreement will become less and less feasible in years to come: information will need to be updated faster than it can be transmitted (latency) and information received from other components can only be trusted conditionally (agency). This is already true of the Web today, even though the risks seem negligible. Nonetheless, nothing ensures that the page being displayed to a user bears any resemblance to the state of the origin resource 'right now.' Web services technologies that suggest computing further results based on out-of-date or untrusted information will only exacerbate these problems.

Ultimately, this will lead to the development of new architectural styles for decentralized systems. We believe that our definition of decentralization is a novel rationale for the adoption of event-based communication; the current client/server orthodoxy will not function under high latency and diverse agency.

Given that our first step is representing decentralized concepts as a series of events, our proposal for DECENT draws upon REST in order to model events as resources; and event notifications as representations. We also expand the role of proxies in REST to model decentralization across agency boundaries using "virtual domains."

We will also implement infrastructure to support DECENT applications, implemented in DECENT style. There are a number of engineering contributions in our application-layer internetworking approach to designing an event router. By reference to the end-to-end principles that worked so well at the network layer for TCP/IP, we expect to contribute a new end-to-end decomposition of traditional Message-Oriented Middleware (MOM) functionality on top of best-effort event notification.

1.4.1. Organization of this proposal

We begin with formal definitions. By introducing the concept of simultaneous agreement in §2, we proceed to define the terms 'centralized,' 'distributed,' and 'decentralized' as simultaneous agreement over name-value pairs.

Defining decentralization as the elimination of simultaneous agreement requirements suggests investigating the limits to simultaneity and agreement, respectively. Therefore, §3 explores latency, which is ultimately a physical limit that forbids absolute simultaneity. Similarly, §4 explores agency, which is the principle that independence implies the right to disagree, which forbids absolute agreement.

The balance of the paper then focuses on how the field of software architecture will be affected by decentralization. §5 considers the historical trends increasing latency and agency, how related research fields have coped, and current avenues (and dead ends) for software architecture.

§6 outlines our response to the challenge of decentralization. §6.1 lays out our argument for new architectural styles expressly designed for decentralized systems, and §6.2 continues with our proposal for a novel kind of event router for assembling decentralized systems.

2. Simultaneous agreement

The term "simultaneous agreement" originated in contract law, specifically for defining financial instruments. An ordinary cash transaction is a simple example of simultaneous agreement upon a trading price, P_{trade}. This represents a (literally) atomic transaction that exchanges money for goods at the same instant (or at least, within the ~100msec threshold of human perception).

In a software architecture using separate components to represent buyer and seller, it is necessary to reify the shared variable as a constraint upon purely local P_{buyer} and P_{seller} variables. Namely, if P_{buyer} equals P_{seller} at some time, then both must also be equal P_{trade} at that time.

The next subsection considers the conditions for feasibility of simultaneous agreement, and if so, how long it takes to achieve. Subsequent subsections consider in turn how P_{trade} is controlled. First, P_{trade} could be centralized under the exclusive control of either the buyer or the seller. Second, P_{trade} could be distributed under the shared control of both parties. Third, we could design the system to work without a single, global, simultaneously agreed P_{trade} at all, which constitutes decentralization.

2.1. Consensus

Lynch, in [Lyn96], claimed "The impossibility of consensus is considered to be one of the most fundamental results of the theory of distributed computing." [FLP85] proved that on a completely asynchronous network (one with maximum message latency d=∞), if even one process can fail, then it is impossible for the remaining processes to come to agreement. [Lyn96] also includes a proof that even with a partially synchronous network with only a finite d and message loss or reordering, consensus requires at least d. In general, tolerating f processes failing requires at least $(f+1)\cdot d$.

Technically, these results are for the consensus problem, which requires each process be initialized with a decision value, and the termination condition is that all processes decide the same value iff the inputs were unanimous. The agreement problem is an equivalent statement where only one process, the leader, proposes a decision value and

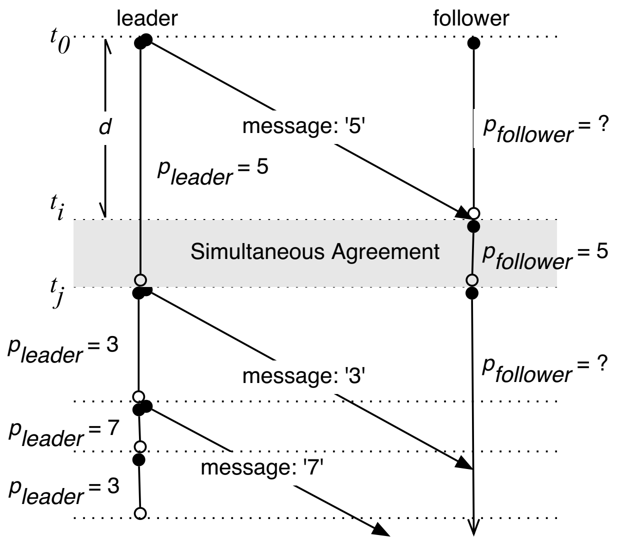

Figure 1. Simultaneous agreement only holds in the shaded region.

all processes exit with the same decision. Mutual exclusion and leader election have also been shown to be equivalent to consensus [Lyn96].

It thus follows that if consensus takes at least d to establish, simultaneous agreement requires holding the leader's value constant for longer than d. Similarly, tolerating f process stopping failures requires holding values constant for longer than $(f+1)\cdot d$. Consequently, it is also impossible to guarantee simultaneous agreement with a leader changing more than $1/d$ times per second (or $1/(f+1)\cdot d$).

To define this condition more precisely, we postulate the existence of a global clock in order to qualify the value of a variable as a time-variate function. It is only possible to synchronize clocks in an inertial frame of reference, so to be sure, the following argument is limited to computers that are at rest with respect to each other. We thus define simultaneous agreement as interval of time satisfying the following conjunction:

$$\exists\ t_j, t_i, t_0 : (t_j \geq t_i) \wedge (t_0 + d \geq t_i) :$$
$$\wedge\ \forall\ v : t_j \geq v \geq t_i : P_{leader}(v) = P_{follower}(v)$$
$$\wedge\ \forall\ u : t_j \geq u \geq t_0 :\ P_{leader}(u) = P_{leader}(t_0)$$

The condition is met in the shaded area of Figure 1, where two world-lines are drawn vertically for the state of the leader and follower processes and the horizontal separation reflects the time it takes a signal to traverse the distance between them. The message takes at most d to travel, and simultaneous agreement only holds for the span of time after the message arrives until P_{leader} changes from 5 to 3. However, if the variable changes after a shorter interval than d, as from 3 to 7, it is impossible for the message to arrive "in time" and hence simultaneous agreement is also impossible.

2.1.1. Leases vs. locks

Note that our definition requires P_{leader} to be constant while agreement is established. At the bottom of the diagram, even though the leader switches back from 7 to 3 before the message "3" arrives at the follower, that is mere coincidence, not simultaneous agreement. Either a follower must contact the leader and request a lock to hold the value constant while it works, or the leader must indicate the period of time it commits to holding a value constant, which is called a lease.

A leased value is represented by a (value, duration) pair. In conjunction with a global clock, it is thus possible for the recipient to determine whether the lease is still valid. In Figure 1, the follower can correctly unset the value of $P_{follower}$ at the first instant the leader is able to change (from 5 to 3) if we extended the contents of the message to specify a lease.

2.1.2. Single-assignment variables

Of course, an architect could avoid disagreement entirely by restating the problem. One example is the technique of single-assignment variables: rather than replacing the value of P_{leader} several times, a series of distinct variables could be set just once: $First\text{-}P_{leader}$, $Second\text{-}P_{leader}$, and so on [Tho94]. On the other hand, this style would also rule out a simple user interface displaying "the price is currently *P*," which is but one example of why software architects rarely choose unfamiliar styles that explicitly model delay and disagreement over shared information.

2.1.3. Agreement over functions

The object of agreement can also be a function, rather than a value. The equivalence seems obvious on a von Neumann architecture computer, where programs are also stored as data, but it requires refining our model of simultaneous agreement slightly.

Specifically, we choose to model named values as a time-varying lookup function. Naming is essential for exchanging information by reference rather than by value.

The semantic of replacing values with references requires a belief that every observer uses the same lookup function. Dereferencing pointers in a common address space is a trivial example of semantic agreement. At the opposite end of the spectrum, determining that a Web page is a valid representation of a resource identifier can be a matter of human judgment.

Agreement over some variable named X between a *leader* and a *follower* separated by latency d is defined as:

$$\forall\ t : t \geq 0:\ Lookup_{follower}(X, t + d) = (Lookup_{leader}(X, t) \vee \varnothing)$$

This definition permits the follower to fail (yield $\varnothing$), such as when the network connection to a remote leader is interrupted. The additional constraint for simultaneity would be similar to §2.1 above.

2.2. Centralization

Centralized (adj): drawn toward a center or brought under the control of a central authority [WN02]

Applied within the context of software architecture, centralization can be described as the practice of assigning exclusive control to modify information or invoke computation.

Two examples of centralized software architecture are the use of a shared database in the client/server style, or the use of a single keyboard input in an event-driven user interface style. Many other components will rely on references to the resulting database record or typed characters, respectively. Nevertheless, those components are the only ones that can modify or generate such information.

A variable X is considered centralized if and only if X is modifiable at only one location, and all other references to X require simultaneous agreement.

By location, we mean something more specific than the street address of a computer. Ultimately, information is represented by arbiters: devices than can make an exclusive choice between representations of information, such as a hole on a punched-card, a magnetic particle on disk, or a flip-flop circuit on a chip (see also §3.1).

If we permit information in a follower arbiter to be used in a computation without establishing simultaneous agreement with the leader arbiter, we contradict the definition of centralization by permitting the same computation at the same time, depending on the same centralized information, to yield different results at different locations.

As we discussed in §2.1, simultaneous agreement comes at a cost. Processor and network latency strictly limit update frequency, which can cause serious problems if a software architect intends that a centralized variable represent some phenomenon in the "real world" that occurs more rapidly.

This is an elementary result of queuing theory, but bears restating: if the request rate exceeds the service rate, total service time increases linearly. Thus, given that d is the maximum latency from the leader to any follower:

No centralized software architecture can correctly react to events occurring more frequently than $1/d$ times per second.

Note that this limit is entirely independent of computing speed or bandwidth; §5 explores its implications.

2.3. Distribution

Distributed (adj): spread out [...] or divided up [WN02]

Sharing control between several agencies requires the concept of a variable that can be modified at several locations, not just one. Therefore, the single leader/multiple follower model of centralization does not apply directly. There are two models for distributing control: either leadership itself can pass from one location to another, or the distributed information is actually derived from an ensemble of centralized variables using a decision function.

Since the former model only has one leader at a time, it ought to be considered ersatz distribution. It is nevertheless popular, in the guise of two-phase commit protocols for 'distributed' databases or as resource management for 'distributed' operating systems. Control passes from one location to another in a round-robin or first-come-first-served manner.

We believe the latter model, using shared decision functions, is a more appropriate choice for modeling how power is distributed in society, or in software. Each individual agent only controls a private preference, but the distributed consensus value is established by a shared decision function, such as the mean, mode, maximum, or minimum. From this viewpoint, Petersen's algorithm for distributed mutual exclusion [Sne93] is only a more complex decision function.

A variable Y is considered distributed if all references to it are the result of applying a decision function *df*() over a set of simultaneously-agreed centralized variables X_1,X_2, …

It follows that since the function *df*()is identical throughout, and its inputs are local variables that are each in simultaneous agreement with remote leaders, then all the local results of *df*() are also in simultaneous agreement with the variable Y.

The minimum time it takes to calculate Y is the maximum latency between *any* two locations in the system. Since simultaneous agreement with a centralized variable requires the maximum delay from the center to any other point, it follows that simultaneous agreement on an ensemble of centralized variables requires at least the maximum delay of each member.

However, actually achieving the lower-bound time of $(f+1)\cdot d$ is more difficult than the centralized case, since it requires strict clock synchronization and phase alignment on every update. Only if every centralized variable changes at the same instant, and for the same lease duration, can we guarantee that all agents have received all updates after only d seconds have elapsed.

The upper bound is the more relevant, and involved, calculation. Without loss of generality, we choose to ignore fault-tolerance (the factor of f). Next, we choose to model the modification of a centralized variable as an event with a lease. Assigning a new value to variable v at time t implies that the value cannot be modified during the interval $[t, t+L_{min})$, where L is the lifetime of the lease. Since it takes d to propagate information, it follows that simultaneous agreement is definitely feasible during the interval $[t+d, t+L_{min})$. That interval is obviously nonempty, since §2.2 above concluded that $L_{min} > d$.

Simultaneous agreement over the distributed variable Y is only possible during the intersection of all the intervals where simultaneous agreement also holds for each centralized variable X_1, X_2, … Without loss of generality, we choose to sort assignment-event times t_1, t_2, … in ascending order. The intersection of several intervals of the form $[t+d, t_i+L_{min})$ from 1 to N is the interval [maximum(minima), minimum(maxima)). Thus, simultaneous agreement over Y holds during:

$$[t_N+d, t_1+L_{min})$$

For this range to be nonempty,

$$L_{min} > (t_N-t_1)+d$$

Since the times are a sorted sequence, the value of (t_N-t_1) can be reduced to the sum of the differences between each successive time:

$$(t_2-t_1) + (t_3-t_2) + \ldots$$

This is at most $(N-1)\cdot\max(t_{k+1}-t_k)$. The crux of our calculation is that it is not possible for successive times to be more than d apart. Informally, we would already be in simultaneous agreement with a variable that didn't change for at least d; we should have included the *prior* modification of that variable in our sort order.

Formally, if $t_{k+1}-t_k \geq d$, then the prior modification time of variable X_{k+1}, called t_{k+1}', is at least $t_{k+1}-L_{min}$ and at most $t_{k+1}-L_{max}$. This implies simultaneous agreement must be possible during the interval $[t_k+\mathrm{d},t_{k+1})$, contradicting the sort order by including t_{k+1}' instead.

Therefore, we conclude that a variable under the distributed control of N agents takes at most $N\cdot d$ to modify.

No N-way distributed software architecture can correctly react to events occurring more frequently than $1/N\cdot d$ times per second.

Note that this limit is also entirely independent of computing speed or bandwidth; §5 explores its implications.

2.4. Decentralization

As discussed in §1.1, decentralization is fundamentally different from either centralization or distribution because it permits multiple simultaneously valid outcomes for the same decision. This assumes that the same name has the same semantics to each observer.

A variable Z is considered decentralized across several agencies A_1, A_2, … if the values (estimates) of local variables A_1's-Z, A_2's-Z, … are all simultaneously valid values of *Z*, even in the absence of simultaneous agreement between agencies.

Returning briefly to a pricing example, the decentralized variable named "yen-dollar conversion rate" is actually represented by several estimates – given the vagaries of network latency – of other bank's best rates. Determining the subset of banks that different traders trust to represent the "yen-dollar conversion rate" is similarly subject to the vagaries of agency conflicts.

Later in this paper, we will present syntax for modeling such conflicts more carefully in the manner of Web resource identifiers (scheme://agency/name) that will reflect the connection between generic, decentralizable concepts and specific, centralized concepts we use in actual computing devices.

Rather than accept the performance limits of simultaneous agreement, we believe decentralized styles of software architecture will rely on estimates of remote information, which will not necessarily be simultaneous (due to latency, §3), nor necessarily agree with local beliefs (due to agency, §4).

The Web's REST architectural style is the inadvertent apotheosis of hybrid decentralization. In a typical three-tier database/application-server/browser architecture, the first two elements are either centralized or distributed within a server cluster, ensuring that at the moment it is generated, at least, a representation is in simultaneous agreement with the abstract resource.

Once that representation leaves the data center, though, agreement is no longer assured – even HTTP 1.1's expiry times are merely advisory. The airline seats your browser blithely displays as available could well have been sold already by the time the page finishes downloading. Only by acquiring an exclusive write lock can the client ensure distributed control (e.g. using WebDAV). §6.1 discusses this and other architectural features we expect to be developed in support of decentralization.

3. Latency

Feynman once observed that "When I talk about everything in the world that is happening 'now,' that does not mean anything... We cannot agree on what 'now' means at a distance." [Fey65]

He was referring to a revolution in physics nearly a century old. Einstein contradicted Newtonian physics by positing that uniform motion in a straight line is relative – that no experiment can reveal which of two observers is moving and which is at rest. The most famous consequence of this principle is that nothing can travel faster than *c*, the speed of light in a vacuum.

Money can buy exponentially increasing computing, communications, and storage capacity, but latency is forever. It is important to understand the sources of latency because it constrains the physically realizable configurations of shared state in a software architecture, or conversely, for estimating the behavior of an application on a given set of machines. The following subsections explain why latency is an absolute limit, how to visualize latency's effect on simultaneous agreement, and how to cope with the uncertainty caused by latency.

3.1. Physical limits

> Interestingly, physical information is apparently, like energy, a *localized* phenomenon. That is, it has a definite location in space, associated with the location of the subsystem whose state is in question. Even information *about* a distant object can be seen as just information in the state of a local object (*e.g.* a memory cell) whose state happens to have become correlated with the state of the distant object through a chain of interactions. Information can be viewed as always flowing locally through space, even in quantum systems [Fra02]

Frank's discussion of the ultimate physical limits to computation, storage, and communication is the key to understanding why latency is an absolute constraint for software architects: abstract information can only be stored by actual arbiters (§2.2). It therefore takes time and energy to transmit information across a channel, if one even exists.

3.1.1. Propagation

As a consequence of relativity, it is not possible for an arbiter, and hence information, to travel faster than *c*. Popular reports of research in quantum computing using entangled particles for "quantum teleportation" and "quantum cryptography" may initially seem eerily "telepathic" (as even Einstein described the instantaneous effects of entanglement), but actual transmission of information requires a corresponding classical message to be sent – below *c* – in order to recover the state [BBC+93].

Practical communications channels actually only achieve small fractions of *c*: photons travel at 67% *c* over fiber; electrons travel at 10% *c* over copper; and atoms travel at 0.0001% *c* over Federal Express. Furthermore, error-correction, compression, and other channel coding techniques also add overhead and jitter to propagation delays.

3.1.2. Bandwidth

In an argument due to [Fra02], information transmission across space can be viewed equivalent to information storage across time, and subject to the same minimum-energy requirements. Since information flux, often termed 'bandwidth,' is also an energy flux, the maximum capacity of channel must be finite. If the desired signal rate is

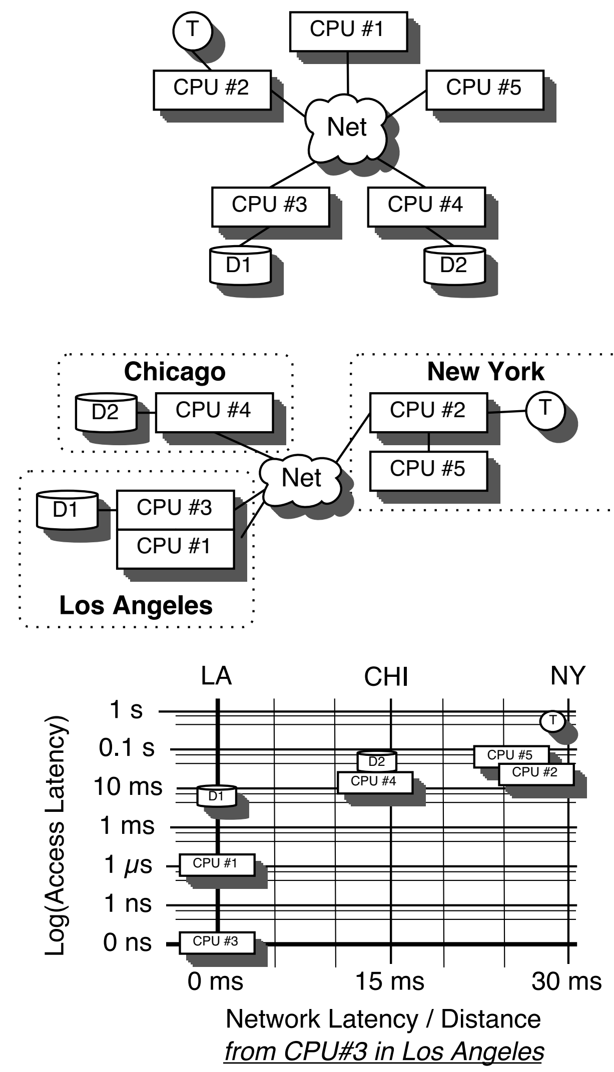

Figure 2. Logical, physical, and latency maps of a computer network

greater than capacity, however, delay increases linearly until buffer space is exhausted and data must be dropped. Furthermore, in store-and-forward networks, on-disk or on-DRAM buffers may be quite large and high bandwidth, but still high-latency, thus adding transit time to and from the buffer to the total propagation delay.

3.1.3. Disconnection

Store-and-forward networks are also robust in the face of limited network partitions. If a line-of-sight is occulted, as for interplanetary networks, or for nomadic cellular users entering a tunnel, the network layer often buffers further to mask the disconnection. System crashes are another source of transient partitions. We choose to model the finite buffer capacity and timeouts for supporting transient disconnection as an additional component of a channel's maximum latency.

3.2. The 'now horizon'

It is possible to map out the components of a software architecture according to the latency of the underlying hardware in order to draw visual conclusions about the feasibility of simultaneous agreement. The key step is transforming ordinary space, where a disk might sit a few inches from a processor, into a gauge of time, where the disk may be tens of milliseconds away, yet another processor in a city hundreds of miles away could still be closer in time.

Using a logarithmic scale, latency visualization is appropriate for modeling everything from components separated by day-long email delays to components running on the same processor separated by microsecond OS thread-switching delays. Interprocess communication (IPC) is merely another kind of delay, indistinguishable from a few extra kilometers of wiring.

Map in hand, note that a circle around an arbiter whose radius is the minimum lease time of the variable traces out a boundary between components that can achieve simultaneous agreement and those that cannot. We term this the 'now horizon,' since it is possible to speak with certainty of the arbiter's value "right now" within it – and impossible beyond it.

This technique is useful for analyzing various hardware configurations, deciding to centralize or distribute control of information, or conversely, determining the maximum update frequency of centralized or distributed variables.

The limitation is that all component latencies have to be measured with respect to a single, central arbiter. It is not generally possible to embed a weighted network graph into two- or three-dimensional space so that all pairwise distances are consistent, even when calculating distance using a different norm (e.g. Manhattan 'city block' distance rather than Euclidean geometry). For example, the entire network in Figure 2 is centered around an arbiter on CPU#3 in Los Angeles, so the 530 msec distance to the tape drive T is correct (30msec to CPU#2 + 500msec to T), as is the 25msec distance to disk D2 (15msec to CPU#4 + 10msec to D2). The "distance" between D2 and T, however, is too short (505 rather than 525).

Nonetheless, this chart is useful enough to tell that a 1Hz variable in LA can still be correctly backed up to stable storage in New York, but a 100Hz variable can't even be used outside of LA.

It is also possible to confirm whether a given component can rely on simultaneous agreement with several variables. If it is within the intersection of multiple now-horizons, it can still establish pairwise SA; two variables can only be jointly consistent only if the minimum lease time of the variable is long enough to completely enclose both arbiters.

3.3. Uncertainty due to latency

Rather than the uncertainty *of* latency from §3.1, consider the uncertainty *due to* latency. Beyond the now-horizon of an arbiter, its current state can only be estimated. The evolution of a variable over time may range from completely random (tossing a coin) to highly predictable (tracking an airliner). [Tou92] discusses these issues in detail, measuring the tendency of estimates to weaken as time passes as 'entropic stability.' Dead reckoning is an example of a technique that exploits it, since the last known position, heading, and time of an airliner could be combined with constraints on its velocity and maneuverability to estimate the region of space it could currently be within.

In general, though, an architect cannot expect predictable estimators based on past information for abstract symbolic variables such as 'part number' or 'sales tax rate.'

4. Agency

While latency is a physical limit, the concept of an agency is socially constructed. It denotes the set of components operating on behalf of a common (human) authority, with the power to establish agreement within the set and disagreement outside it. The anthropocentrism of the concept may seem irrelevant to the concerns of software architecture, but the missing link is the tacit assumption that every computing device is owned by a person or organization, and is expected to operate on behalf of that agency.

Our department recently installed a virus scanner on our mail server that promptly deleted several drafts of this paper. A message's *infected-p* flag is set solely by the scanner, which is configured solely by the support staff, and no feature of the centralized scanner software is prepared to brook dissent from its judgment (furthermore, the attachments are deleted forthwith, in accordance with a separate privacy policy that forbids storing mail!) In lieu of a decentralized virus scanner that lets each agency – sender, receiver, administrator – determine *infected-p* locally and react independently, we defeated the software entirely by controlling a variable that was under our control: the attachment filename, the truth of which the scanner entrusts the sender's agency as blithely as it arrogated exclusive control of *infected-p*.

Coordinating multiple agencies is arguably closer to the political economic definition of the term 'decentralization' than the effects of high latency alone. The following subsections explore the basis of agency limits, a rule to enumerate the extent of an agency, and ways to cope with the uncertainty caused by inter-agency conflicts.

4.1. Philosophical limits

Philosophically, an agency is a set of trust relationships between components, legitimized by indirect delegation from human agents' authority. An agency maybe entrusted to authorize a given username/password combination to invoke services, or inherit its owner's processor priority level, for example.

A message attributed to another computer is merely a string of bytes, but a message attributed to another agency is an assertion to be judged, a statement of opinion. The boundary between agencies is that another agency's assertions must not be accepted automatically as a local belief. While protocols exist to enable agreements between agencies, it must be voluntary if the independence of decentralized agencies is to mean anything.

As an illustration, agencies maintain their own ontology. The ASCII character set is a simple system of meaning that was endorsed by a series of more and more broadly-based standards organizations, reflecting the authority of a single company (IBM) at first, to the point that today many computing devices cannot be configured to use any other symbols for input/output. And yet, two instances of identical software components written by the same vendor, but running on both an American and a Japanese company's equipment may have to renegotiate even this basic level of meaning to before establishing agreement over higher-level concepts like 'Customer Name.'

4.2. Agency boundaries

Determining agency membership of a component is merely the inverse of the independence principle above: any component that unconditionally agrees with all the assertions of an agency is part of it.

An atomic transaction is an excellent example. A customer depositing money in a bank account may have her own opinion of the current account balance, but to commit a transaction, she must agree with the bank's estimate or there is no deal (in software; real-world legal remedies may be in order quite apart from the software architecture's affordances). Insofar as that transaction was concerned, she was within the Bank's agency boundary, unconditionally accepting the Bank's control of her id, password, balance, and transaction state.

On the other hand, defining whether an email contains a virus or constitutes spam is in the eye of the beholder. No amount of unsolicited email from us ought to convince your email system to unconditionally accept our assertion that our message is virus-free. You might inspect it yourself, or you might trust us for other reasons, but the fact that our *infected-p* flag is stored, believed, and acted upon by every computer we own is no reason to assign our flag value to your agency's own *infected-p* state.

Assignment requires agreement, and components within an agency boundary agree with each other by default just as those components must be able to disagree with other agencies' components by default.

4.3. Uncertainty due to agency

As with latency in §3.3, there may uncertainty *of* a component's agency obligations with respect to a given piece of information (deception), but there is uncertainty of a different order arising *due to* crossing an agency boundary (confusion). A bid message from "Bob" without knowing the agency making that assertion leaves us uncertain of which agency's "Bob" this is. If we compare two bids for a "2-day" car rental without the agency responsible for uttering "day", we cannot compare the precise timespan (24 hours from now? Now until noon tomorrow?). The most direct example of this kind of uncertainty is public key management itself: establishing an agreement that a key corresponds to an identity depends crucially on each agency legitimizing each assertion in the chain.

5. Coping with latency & agency limits

The challenge of integrating software packaged as components has been thoroughly explored over the last two decades. [Ore98] surveyed the challenges of integrating software developed by several different organizations, a process termed "Decentralized Software Evolution." It could be said the next challenge will be integrating software delivered as a service over the network instead.

Relying on network services across latency and agency boundaries requires designing solutions that minimize the need for simultaneous agreement. This section applies our definitions to examine the historical trends that are increasing latency and agency; discuss how other computing disciplines are coping with decentralization; and evaluate early responses from software architecture research.

5.1. Trends increasing latency & agency

Moore's Law is no match for Einstein's. In coming decades, microprocessors are generally expected to continue exponential decreases in feature size (Moore's Law); exponential decreases in cycle time; and exponential increases in total price/performance (Joy's Law). Peak bandwidth across all media – copper, fiber, and wireless – is expected to grow at an even faster rate.

We also expect new kinds of networks to emerge, such as wireless, ad hoc personal-, neighborhood-, and interplanetary-scale connectivity. Perversely, these innovations portend higher latency, lower bandwidth and more transient connectivity in exchange for more ubiquitous reach.

Taken together, while latency can be expected to improve, approaching its fundamental limits as technologies for all-optical networking become available, the relative opportunity cost of a millisecond's latency measured in terms of foregone computing power and bandwidth•delay product will be increasing substantially.

We also expect the current trend for increasing interorganizational integration to continue apace. Known variously as "eBusiness" or "real-time enterprise" initiatives, as more of the economy's business processes come online, we can expect the velocity of information throughout the supply chain to increase. Software architects will be expected to model higher-frequency variables as markets of all kinds accelerate and perhaps even increase in volatility.

5.2. Related approaches to decentralization

Combined, these trends promise to shrink the now horizon and increase the diversity of agencies simultaneously. Coping with these shifts will require designing for decentralization, which suggests reviewing similar research on other aspects of computing: microprocessors, internetworking, and collaboration software.

5.2.1. 'Clockless' chips

Conventional microprocessors require simultaneous agreement throughout the chip on each instruction cycle. The clock signal required to synchronize today's dissipates nearly a third of the total power budget [Sut02]. Considering that a 1GHz processor has a switching time of 1 nsec, and electrons travel at just 10% *c*, the now horizon of the entire chip is only an inch! Eventually, we will have registers capable of switching faster than a signal can reach the rest of the chip. One conservative response is called heterogeneous clocking, which divides the chip into several high-frequency subcircuits connected by a miniature LAN.

A more creative response has been eliminating the requirement for chip-wide simultaneous agreement using delay-insensitive circuits. So-called asynchronous microprocessors have already realized significant decreases in power consumption and increases in overall performance.

5.2.2. Best-effort packet switching

Back when ARPAnet was still a risky experiment, conventional circuit- switched networks took advantage of explicit signaling, allowing both ends of a connection to agree on the capacity and availability of a link separately from the data flow. The boldest claim of packet-switching was that congestion and link failure did not need to be explicitly signaled. By decentralizing control of a link, an end-to-end overlay protocol, TCP, could still estimate the link capacity using a sliding-window protocol rather than requiring simultaneous agreement on buffer states.

The essential premise of internetworking is to network networks owned by separate agencies, making it a great success for decentralized architecture. And yet, even TCP/IP requires too much agreement for very high bandwidth·delay links, so [Tou92] introduced a branching-window protocol that uses "extra" bandwidth to reduce effective latency by sending speculative information based on estimates of likely future states at the receiver.

5.2.3. Real-time collaborative authoring

High latency between multiple participants made simultaneous agreement over a shared document too sluggish for real-time collaborative work. In order to decentralize the "distributed" document beyond the now horizon, [EG89] introduced the technique of distributed operational transforms (dOPT). Roughly speaking, local document transformation events are broadcast along with a priority, which then is used to reconstruct ordered subsets to "replay" at the other locations.

5.3. Identifying decentralized software

Decentralization is an active research topic in software today. The term has been used, overbroadly, to describe a wide range of "P2P" systems.

"Peer-to-peer" has become a popular phrase in the wake of several successful, innovative applications that share in common little more than *not* being standard client/server architectures [MKL+02]. Under the P2P banner, we find Napster, which queried a centralized song database; Freenet, which is more nearly decentralized, but at the expense of searchability; instant messaging products that enable user-to-user communication, but only through completely centralized servers; and a host of distributed computing projects that leverage decentralization only insofar as separate agencies own contributed computing resources (the actual applications run on a grid, say, are still client/server).

We believe purely decentralized software development is rare today, outside of medical, military, and financial applications. Within computer science proper, operating system design – specifically, interprocess isolation – is a miniature recapitulation of the challenge of decentralization. Programs are owned and operated on behalf of distinct users, and user-mode programs must be able to function even with very high latency between processor time slices.

6. Decentralizing Software Architecture

The common denominator of the approaches for coping with decentralization in §5.2 is that they model interaction as *events*. Rather than relying on a clock for moving data across a chip in lockstep, asynchronous circuits pass control to the next state by handshaking signals, which are analyzed as event traces. Rather than depend on explicit signaling, TCP re-estimates the window size in the event an IP packet is dropped. Rather than slow down the pace of collaboration to the slowest participant's, operational transforms permit real-time editing as fast as latency permits by reasoning about consistent, partial event sequences.

Without simultaneous agreement – whether caused by short leases, long latency, or agency conflicts – it is impossible to speak of the "here and now," only to reconstruct the past from events generated "there and then." This section introduces our proposals for decentralizing software architecture: a novel event-based architectural style, and a novel event router design.

6.1. The DECENT architectural style

> "The components of a loosely coupled systems are designed to generate and respond to asynchronous events" [CRW01]

Eliminating requirements for simultaneous agreement between components is a fundamental motivation for event-based architectural styles. As noted above, many responses to decentralization begin by modeling interactions with remote states as events. Event-based interaction also more directly models the underlying physical reality: communications are one-way, delayed, and unreliable.

Event-based architectural styles continue to function beyond the now horizon, where synchronized styles such as client/server with locked resources are forced to insert wait states to accommodate high-latency clients. This is a novel case for event-based integration; under circumstances where simultaneous agreement is feasible, it has remained a matter of convenience between push- vs. pull- transmission or topics vs. queue naming whether to recommend a message-based or event-based architectural style. But once shared state becomes infeasible, event models have the advantage of continuing to function correctly as delay increases.

The second force motivating decentralization is agency conflict, for which we introduce the concept of resource and representation from the Web's REpresentational State Transfer (REST) style [Fie00]. URLs combine a host portion, for tracking the long-lived identity of an agency, typically a DNS name, with a pathname portion for indicating which resource at an agency is to be represented.

By modeling event sources as resources, we inherit abstractions for sharing identifiers used for subscriptions or filters. By modeling event notifications as time-variate representations of a resource, we inherit naming, access control, cache validity, and a marshalling format for representing arbitrary media types. But most importantly, it allows us to reason about a sequence of event representations in ways that isolated messages do not permit. A newer representation for the same resource is considered a replacement, allowing a connector to automatically discard representations arriving more rapidly than can be delivered. A representation can also expire, wherever it has reached in the network. A representation has a unique entity tag permitting routing loops to be detected and prevented, which is essential in a decentralized event notification service without a simultaneously-agreed routing table (subscriber list).

The other precedent DECENT builds upon is the C2 style, which characterizes components solely in terms of their notification (input) and request (output) events. Current developments in Web services technology fit this model directly. We intend to extend the C2 definition of component to capture agency aspects, and to extend the C2 definition of connectors to capture latency expectations, for analysis and tuning in the future.

The major departure we envision, though, is generalizing C2's layered lattice of components assembled by an architect at design- or run-time, to an arbitrary network of services that can be extended by any authorized user at any time. The subscription list is the routing table, which is also the architectural configuration of how components are connected.

In a network model, it becomes natural to reason about components, partitions, and routes; if a part of the architecture requires encryption and authentication for access, it suffices to isolate that component and force all traffic to flow across a single route, and thus through the security filters ('firewall').

6.2. A DECENT event router

All this leads to a complementary investigation: how to implement a DECENT-style application. Clearly, this calls for an event notification service, which is typically abstracted as a bus that can select a subset of matching messages from a set and deliver them, ideally asynchronously to reduce notification latency [RW97]. Rather than approximating the ideal of a single event notification service connecting all components of an architecture, we plan on implementing a decentralized event router, without assuming shared information about topics, subscribers, or events. Our model is an HTTP server, which is designed to support a decentralized web of content by serving up a portion of the global URL namespace without any *a priori* coordination with any other Web servers, clients, or proxies. We have prior commercial experience designing event routers for KnowNow, Inc. that proved significantly easier to install, administer, and develop with than competitive systems by virtue of deferring to Web practices whenever possible. The following subsections highlight three aspects of our design that appeal to precedents set by the Internet.

6.2.1. ALIN

Just as IP routers enabled interconnection of multiple network-layer protocols, we believe an application-layer router can enable interconnection of services currently limited to access by email (SMTP), file transfer (FTP), the Web (HTTP), as well as proprietary interconnects such as AOL Instant Messenger [Day00] and emerging conventions such as peered WebDAV repositories.

The SOAP and SOAP-Routing specifications [Gud02] reflect similar thinking insofar as SOAP 1.2 is actually independent of any particular application-layer transfer protocol. There are bindings to HTTP and SMTP, but conceivably any one-way, 8-bit clean, ordered channel could be used. In a sense, SOAP is a toolkit for generating new application-layer protocols, increasing the need for a router.

6.2.2. MCP/EP

Just as TCP provided reliable, ordered messaging on top of IP's unreliable, unordered primitives as an end-to-end service – a TCP stack running on smart hosts, leaving the internals dumb (and fast) – we believe traditional Message-Oriented Middleware (MOM) communication patterns are worth supporting, but not at the expense of complicating the core. Specifically, transactional messaging, queuing, presence, and load balancing can all be implemented as event-notification patterns without router support.

6.2.3. Striping

Just as RAID provided reliable, available, and scalable file service layered atop arrays of unreliable, inexpensive, isolated disks, we plan to provide reliable, available, and scalable event routing capacity out of unreliable, inexpensive, and isolated servers. But while the key to RAID was striping across a fixed set of disks, the key to maintaining a very large, dynamic, interorganizational cluster of routers is a more clever striping function known as a consistent hash [KL+97]. By posting events to multiple routers, and subscribing to multiple routers, we plan to provide end-to-end, user-defined levels of reliability and availability using a K-of-N failure model.

7. Conclusion

Decentralization is a familiar phenomenon in human affairs, and will soon affect the architecture of software systems that cross organizational boundaries as well. When it is not accounted for, phenomena such as spam result: the validity of a message is in the eye of the beholder, yet our distributed email software cannot distinguish separate agencies.

Latency and agency are two forces driving the decentralization of software architecture. By defining centralization as equivalent to simultaneous agreement, we argued that absolute limits to both simultaneity – latency – and to agreement – agency – are likely to motivate the development of decentralized software architectures.

Our own proposal for an architectural style supporting decentralization begins by explicitly modeling latency and agency as aspects of event-based communications. Specifically, we claim that DECentralized Event Notification Transfer (DECENT) is an extension of concepts from the C2 and REpresentational State Transfer (REST) styles that enables 3rd- and even 4th-party extensibility of deployed services. The key technical insight justifying this claim is that software packaged as network services, unlike prior generations of component packaging technology, reflects the same connectivity challenges at the application layer as IP internetworking did at the network layer decades earlier.

Our resulting proposal for an event router for Application-Layer InterNetworking (ALIN) is an innovative infrastructure for event-based communication that divides the traditional function of several MOM facilities into a basic Event Protocol layer and an end-to-end Message Control Protocol layer above it.

Since latency and agency are absolute limits, independent of improvements in computing or communications capacity, we expect these two concerns will dominate the organization of future planetary-scale software integration. As architects tend towards modeling higher-frequency phenomena resulting from interorganizational integration and automation, we believe decentralized software architectures will prove more effective than today's popular centralized styles such as client/server.

8. Acknowledgements

The author gratefully acknowledges the assistance of Dr. Richard N. Taylor, Dr. Joseph Touch, Dr. E. James Whitehead, Dr. Roy Fielding, Eric M. Dashofy and Adam F. Rifkin.

Effort partially sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0599. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Laboratory, or the U.S. Government.

9. References

[ACM98] Association for Computing Machinery, "ACM Computing Classification System (1998)", New York, NY. http://www.acm.org/class/1998/ccs98.html

[BBC+93] Bennett, C. H., Brassard, G., Crepeau, C., Jozsa, R., Peres, A., and Wootters, W., "Teleporting an Unknown Quantum State via Dual Classical and Einstein-Podolsky-Rosen Channels," *Phys. Rev. Lett.* 70, 1895 (1993).

[BC92] @@Berners-Lee, T. , Caillau, R., et al. CACM paper. Alt: Caillau+Gilles, *how the web was born*

[CRW01] Carzaniga, A., Rosenblum, D. S., Wolf, A. L., "Design and Evaluation of a Wide-Area Event Notification Service, ACM *Transactions on Computer Systems*, Vol. 19, No. 3, August 2001, pp332–383.

[Day00] Day, M., Aggarwal,S., Mohr, G., et al. *RFC 2779: Instant Messaging / Presence Protocol Requirements.* IETF, February 2000.

[EG89] Ellis, C. and Gibbs, S., "Concurrency Control in Groupware Systems," in ACM SIGMOD '89, pp. 399-407.

[Fey65] Feynman, R., *The Character of Physical Law* (1964 Messenger Lectures at Cornell), MIT Press, Cambridge, MA, 1965, p93.

[Fie00] Fielding, R. T. *Architectural Styles and the Design of Network-based Software Architectures*, Ph.D. Dissertation, University of California at Irvine, 2000.

[FLP85] Fisher, M. J., Lynch, N. A., Paterson, M. S., "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, 32(2):374-382, April 1985.

[Fra02] Frank, Michael P. "Physical Limits of Computing," IEEE *Computing in Science & Engineering*, May/June 2002.

[Gud02] Gudgin, M., Hadley, M., Mendelsohn, N., et al. *SOAP Version 1.2 Part 1: Messaging Framework*, World Wide Web Consortium, December 2002

[KL+97] Karger, D. R., Lehman, E., Leighton, F. T., Panigrahy, R., Levine, M. S., Lewin, D., "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web, " *STOC* 1997, pp654-663

[Lyn96] Lynch, N. A. *Distributed Algorithms*, Morgan Kaufmann, San Francisco, 1996. p. 371, pp. 798-810.

[MKL+02] Milojicic, D. S., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard, B., Rollins, S., and Xu, Z. *Peer-to-Peer Computing*, HPL-2002-57, HP Labs, Palo Alto, 2002

[Ore98] Oreizy, P. “Decentralized Software Evolution,” in *Proceedings of the International Conference on the Principles of Software Evolution*. Kyoto, Japan. April 20-21, 1998.

[RW97] Rosenblum, D. S. and Wolf, A.L., “A Design Framework for Internet-Scale Event Observation and Notification,” in *ESEC / SIGSOFT FSE* 1997, pp. 344-360

[Sne93] Van De Snepscheut, J. L. A. *What Computing Is All About*, Springer Verlag, 1993

[Sut02] Sutherland, I. E. and Ebergen, J., “Computers Without Clocks,” *Scientific American*, Aug. 2002.

[Tho94] Thornley, J. *A Parallel Programming Model with Sequential Semantics*, Ph.D. Dissertation, Calif. Inst. of Technology, May 1994

[Tou92] Touch, J. D. *Mirage: A Model for Latency in Communication*, Ph.D. Dissertation, Univ. of Penn, Jan. 1992.

[WN02] Miller, G. A., et al., WordNet version 1.7.