Pipeline
Knowledge Is Power

www.pipelinepub.com Volume 4, Issue 4

Self-* Networks: Helping Networks Help Themselves

by Wedge Greene

Self-* (self-star) Systems

Continuing our discussion on Autonomic Networks and Autonomic Communications, we dive into designing and building what we call **self-*** (pronounced *self-star*) systems. Self-* is a shortcut term for systems which are designed specifically to be self-organizing and self-managing, including properties such as: self-defining, self-configuring, self-awareness, self-optimizing, self-protecting, self-healing (self-monitoring, self-diagnostics, self-restoration). That is a big-bite! Further, these systems exhibit the structural characteristic of self similarity. Unfortunately, they are not spontaneously generating.

Asked about self-healing in systems and networks, Dan Druta of AT&T returned a lengthy and emphatic plea:

> "The issue of self-healing distributed computing systems is like finding [a] cure for cancer. We have a vague idea why it might hit and can take some precautions but when it hits the body there's no real cure. Money, energy and endless efforts are being put in palliative treatments to basically patch the "system." Our body is a system and, while not that distributed and not perfect, is fairly resilient and quite autonomic. I had to draw this parallel because some times the desired architecture is in front of us. We just need to wear the right glasses to see it. What's the key? I guess it boils down to three fundamental functions: monitoring, redundancy, and central control."

Fear, Uncertainty and Doubt (FUD)

> H. L. Mencken: "For every complex problem, there is an answer that is clear, simple--and wrong".

Rudy Puryear, who heads IT for Bain in the Americas, complains that important change in our IT systems and networks is not happening because "a significant amount of IT spending [is] for keeping the lights on, managing the complexity. That squeezes out dollars for spending on innovation. We have observed that in most organizations, 70 to 90 percent or more of the IT spend is locked up in depreciation, business as usual, "keeping the lights on," and managing the complexity, often leaving a very small portion of the spend that can truly make a difference in

business performance." [CIO] Unfortunately, corporations cannot break out of this trap unless we significantly alter the characteristics of IT systems to directly attack and vanquish this burden of support costs, but we have a *catch-22*: Significant money needs to be directed toward innovation in order to remove this burden of support costs, but the pressures on that leftover 10-30% of the IT budget are legion – every sub-group in the organization is fighting for that capital.

Past profligate spending in IT and networks has resulted in companies placing strict budgets on IT groups and in turn has passed significant control over spending to external purchasing departments. This was designed to control spending in the technology sector and direct what spending occurred to clear business needs and initiatives. This is a good goal. Unfortunately, these organizational controls often get warped by the dynamics of bureaucratic corporations. Spending decisions get captured by managers who have no understanding of the rationale for selection, little direct contact with the groups, and no experience with the problems for which the money is allocated. Too often, reasons for choosing the vendor have little to do with the services and products acquired. For one example, we all know the nightmare stories of purchasing blocking the proposed partner organization because it is not on the list; and will never be allowed to get on the list. Smaller groups and start-up organizations, like the majority of those supporting self-*, have a particularly hard time with this. Big corporations can better target the specific drivers of procurement which only indirectly reflect on the problem the purchase is directed at.

Ironically, this trend has resulted in significant, and perhaps proportionally too much, IT capital being spent on procurement software and systems. When a purchasing department buys enforcement software and deploys this to all departments, capturing a significant part of the free IT capital, something is wrong with the priorities of management. After all, no company is about purchasing things – the point of a business is to deliver products and services to a customer.

For this constricted, uncommitted capital budget, competitive pressure is extreme; every group is fighting for its new facility and every approved vendor company is fighting to supply it. Big established vendors have significant structural advantage with better funded and larger sales and marketing organizations. Self-* companies, mostly startups, frequently fumble in this area – no one yet knows how to frame the successful marketing pitch for self-*. But, additionally, the reality is that negative market forces have come to dominate the IT market. Disinformation is being disseminated and spin predominates over fact. The truth is that spreading Fear, Uncertainty, and Doubt (FUD) works; It works to prolong the status quo against novel and more efficient methods. Today's market is choked with misinformation on Self-* products leading purchaser into FUD.

While the Long tail effect is changing the market for consumer goods, it does not seem to play a role in IT. Vested vendors, who are already listed partners by procurement, have an interest in keeping things as they are. Big vendor corporations with mature products want to keep their market dominance. So every time a self-* company claims to better meet the core business needs (for example: reliability, agile, survivable, significantly cost reducing), the established companies claim their software already does this. In fact it is their marketing departments copying self-* arguments and claiming unwarranted product properties, and not the development groups of these companies copying facility. So management is beset

by everyone claiming to be, for instance, agile, and the value of "agile" is depreciated. This is a successful block to the introduction or real new features for software and a barrier for entry of new paradigms of computing.

But there are barriers even deeper in the corporation. Many developers and designers have desires to keep things as they are. The designer gains reputation for creating a structure to solve a problem; the developer gains reputation for completing a working software artifact or network device. If the designer is told his approach is wrong, if the developer must be trained in a new approach – losing productivity and status, neither will want to embrace change. New approaches are hard to grasp – the classic paradigm change problem. While these paradigm changes occur [database centric systems, structured programming, objected-oriented design, application servers, web-centric applications, services/SOA/SaaS], the adoption is slow. Christenson has shown that, in order to flip this core community toward new solutions, executives must go to significant lengths to foster a culture of change, among them targeting incentives to real business needs.

Paremus CEO and Founder, Richard Nicholson:

> "Whilst the technology industry is well versed with the robust, agile, cost effect marketing messages that must be associated with each and every product launch or new Industry trend, the reality is that the industry has collectively failed to deliver these benefits. Unnecessary environmental complexity has flourished. Paremus believes that a fundamental part of the problem is that the IT industry does not understand how to architect the type of adaptive distributed system demand by modern business. The irony is that such systems are common place and surround us. Biological, political, and social systems all can be classified as "Complex Adaptive Systems" (CAS). Such systems are highly adaptive, extremely robust to component failure, and extremely efficient. The characteristics of CAS systems, including hierarchical dynamic assembly and population self-control (Stigmergy) are well understood, and provide the architectural blue-prints for the next generation of distributed software system."

Research & Forums

As we mentioned last month, there are many current public efforts. Here is how they characterize the work of defining autonomic communications. Here we briefly look at what each does in their own words.

The **Task Force on Autonomous and Autonomic Systems** (**TFAAS**), "was established to address research issues concerned with creating self-directing and self-managing systems (*selfware* or *self-** properties). The overarching vision of AAS is the creation of self-directing and self-managing systems in accordance with high-level guidance from humans to address today's concerns of complexity and total cost of ownership while meeting tomorrow's needs for pervasive and ubiquitous computation and communication." "The TFAAS … focuses on specifically addressing architectures, frameworks, paradigms, components, tools, environments, languages, applications and lessons for Autonomous and Autonomic Computing and Communications (Systems)."

Autonomic Communication Forum (ACForum): "The ACF initiative is founded on the belief that a radical paradigm shift towards a self-organizing, self-managing, and context-aware autonomous network, considered in a technological, social, and economic context, is the only adequate response to the increasingly high complexity and demands now being placed on the Internet." The ACForum runs conferences and sponsors academic journal articles.

Autonomic Network Architecture (ANA) is an EU funded 4 year project under the Future Emerging Technologies program: "ANA takes on the challenge of not only producing original scientific research results and a novel architectural design, but also showing that they work in real situations, and using the experience gained experimentally as feedback to refine the architectural models and other research results." ANA seeks "to identify fundamental autonomic networking principles for an evolving network which includes the various self-x attributes essential to autonomic communication such as self-management, self-optimization, self-monitoring, self-repair, and self-protection." They expect to produce an architecture "that enables flexible, dynamic and fully autonomic formation of large-scale networks in which the

functionalities of each constituent network node are also composed in an autonomic fashion. This architecture must allow dynamic adaptation and re-organization of the network according to the working, economical and social needs of the users. Moreover, it must support mobile nodes and multiple administrative domains. ” Once this architecture and the fundamental principles are established, ANA will build a working prototype network as a proof-of-concept lab.

CASCADAS: (Component-ware for Autonomic Situation-aware Communications and Dynamically Adaptable Services) has been funded by the EU as a “blue skies” research project involving 14 universities, organizations, and service providers. “The aim of CASCADAS is to develop networks which can – to an extent - think for themselves. Ideally, the systems could run themselves better with little human influence. This is of great interest to business, as it offers lower-cost services in computing and networks,” says Maurice Mulvenna, a senior lecturer on computer science at the University of Ulster's School of Computing and Mathematics. [Foyle News, 11/1/2006]

> “CASCADAS studies how strategic needs impact on future communication paradigms. The central objective of CASCADAS is to identify, develop, and evaluate a general-purpose abstraction for autonomic communication services, in which components autonomously achieve self-organization and self-adaptation towards the provision of adaptive and situated communication-intensive services. The scientific principles of situation awareness, semantic self-organization, self-*, similarity, and autonomic component-ware guide the project.” Simply put, the goal of Autonomic Networks is explained by Maurice Mulvenna: “What we will be exploring is how to make networks aware of what they are carrying and to make decisions based on that information.”
>
> “CASCADAS considers a scenario in which dynamic and heterogeneous networks, possibly enriched with sensors and devices connecting with the physical world, have to host the dynamic deployment and execution of applications and services. Such applications and services have to serve users according to both their social situation and the current network and physical situations.”

Serenity is a recent R&D activity again funded by the EU. Within 3 years, Serenity is expected to produce an architecture for dependability and security in computers. 15 organizations are contributing to the project with Nokia and SAP principle industrial partners as well as Italy’s Deep Blue. Results in little more than a year of activity are rather impressive.

> “Ambient intelligence is the idea that people will be surrounded by intelligent and intuitive interfaces embedded in everyday objects around us and an environment recognizing and responding to the presence of individuals in an invisible way. It builds on three major concepts: ubiquitous computing, ubiquitous communication and intelligent user interfaces.”
>
> Serenity is testing its architecture against specific reference scenarios provided by the major industrial partners. Among these scenario domains is ‘QoS in mobile communications over ubiquitous networks’ and ‘mesh networks of smart items and sensors.’

BIONETS (BIOlogically inspired NETwork and Services) is seeking to develop the architecture of communications and networks which will support millions and billions of small sensors and mobile devices – as such it is seeking self-managed systems for the Ubiquitous Computing and Pervasive Networks. BIONETS is sponsored by the European Commission. Several service providers participate as project partners. BIONETS sponsors this year's Autonomics 2007 conference (www.autonomics.org) as well as bio-inspired methodologies and tools (www.bionetics.org).

Lastly, there is the **Open Grid Forum** (OGF), a union of the Global Grid Forum and the EGA. Being a forum, membership is open. It includes an industry interest group for telecommunications. Periodic meetings provide a conference format for exploring standards progress and deployment experiences. "The long-term vision of Grid can be summed up as follows: "Scalable distributed computing across multiple heterogeneous platforms, locations, and organizations. Grid is composed from the following characteristics and goals:

- Management of Virtualized Infrastructure
- Resource pooling and sharing
- Self-monitoring and improvement
- Dynamic resource provisioning"

What part of nature actually helps in purposeful systems design?

Common to most of these groups are biological systems as models for building autonomic communications. We understand that these bio-models, and the organizations engaged in research in their use, can appear to be a little pie-in-the-sky. Much of the research in application of nature-based designs to networks and information systems is very abstract and currently out of reach of real problems and solutions. However, nature is still useful as analogy and a source of inspirations for application patterns.

Fundamentally, building networks and applications is still a matter of design, not evolution or chance. One concludes that autonomic communications, as idealistically

expressed above, is still out of reach. Today we must concentrate on patterns which reflect the *purposeful* design of networks, services, and support systems. Fortunately, architecting and coding self configuring, self-healing systems is not only possible; it's been done - again and again by different groups.

In part one (last month's), *Autonomic Networks*, we comment: "Nature is not designed. All the interactions have been worked out by minute changes in the activities of the individual species acting over a long change. Ecosystems have no team with the job of network designer creating the pretty schematic…" We however have the possibility of introducing active design. These designed patterns are then fed into what is actually an existing complex system of network elements, connections, services, and support systems. The key is that the stimuli and the micro behavior driving the complete system can be purposefully designed. Further, like a deity, we can observe these systems from the outside, see what works as we wish, and replicate that successful pattern to other parts of the systems environment. So it is a three step approach,

(1) Design stimuli and controls;

(2) Deploy & watch how these affect the complex system, and

(3) Reproduce & disseminate the patterns which work.

The use of stimuli closely follows a basic observation of complex systems in nature.

> **"Stigmergy** is a method of indirect communication in a self-organizing emergent system where its individual parts communicate with one another by modifying their local environment."
>
> "The term [stigmergy] is also employed in experimental research in robotics, multi-agent systems and communication in computer networks. In these fields there exist two types of stigmergy: active and passive. The first kind occurs when a robotic or otherwise intelligent "agent" alters its environment so as to affect the sensory input of another agent. The second occurs when an agent's action alters its environment such that the environmental changes made by a different agent are also modified. A typical example of

active stigmergy is leaving behind artifacts for others to pick up or follow. An example of passive stigmergy is when agent-A tries to remove all artifacts from a container, while agent-B tries to fill the container completely." [Wikipedia]

This indirect control is fundamental to how self-* systems are built today.

Architecture of Self-* Systems

Returning again to Dan Druta of AT&T, he clearly desires some core basics features of self-healing systems:

"… the 3 key functions:

1. Monitoring - all the nodes in the system or network must be instrumented… Like fractals, any node of a system might and I would say IS a system in itself so the same rules can be recursively applied down.
2. Redundancy - to have a self healing system redundancy is mandatory. Call it fault tolerance, call it cluster, call it grid; it must be there.
3. Central Control - the spine of the network or system has to be able to analyze, correlate and take decisions based on n-factor situations. Now, where existing architectures fail is in the ability to automate these central control tasks, and apply recovery rules with minimal human intervention."

We see these as starting points toward control systems and patterns which embody situational awareness, self-similarity, self-*, and semantic self-organization.

There are commonalities in the approaches to designing self-* systems. These originate from the use of Internet Middleware. Most self-* design patterns liberally use the services developed for control and communication in the Internet – often extending and improving these. Principle of these is the Directory. This has been extended to the logical construct of the Registry. Originally, LDAP directories were the implementation mechanism, but databases behind a service interface can also be used. The Registry is a semantic repository of meta-data and collaborative patterns. The Registry stores the pattern of service deployment and the initial state (startup data) for services. Think of the Registry as a static model of the entire self-* system. It can store references to all the hardware (called platforms) and the information necessary to securely log into these systems and deploy services on them. It stores the structure of logical domains in the system. It stores the services needed in each domain. It stores the service dependency mix for an application so it can be maintained as a unit. It stores the start state for any service which is launched.

Most self-* systems use the same pattern of services to both launch and maintain the health of services. A collection of interacting services components is called an application pattern. The assumption is any specific service component might (in fact, will at some point) fail, and when discovered, the system will regenerate this service on a healthy platform. All self-* systems are distributed systems, and assume there is a pool of resources on which to deploy services. Usually today, this pool is called a Grid and consists of many servers of similar (but not identical) characteristics. So, for example, if a specific server fails or becomes overloaded, the

monitoring component discovers this and then regenerates all the lost services on healthy servers from the grid pool.

In self-* systems, usually there are embedded behavioral constraints on services. A service must not assume anything about the platform on which it is to be deployed. Instead, the service is deployed into a virtual machine (or virtual grid middleware, such as a specialized application server or web server) that is deployed on every server in the grid pool. This virtual machine is usually called a Container. Services understand the Container artifact and the Container can host any service. A Service Loader or Service Launcher looks at the Registry for the pattern of service deployment and then launches the services into containers to match this pattern. The service code must come from some form of code server. In Java systems this is usually an HTPP server delivering Java Jar files. So deploying a software update just becomes a matter of putting the new version of code in a code server and revoking the old code's authentication. The updates are then distributed throughout the system.

Generally, services must contain an interface to the Registry that will fetch their initial state, or an interface to a proxy service which passes them this information. Getting and using this data, the service loads and starts up. Services also are required to contain a management/control interface. This interface finds and actively links to a monitoring service – it registers with the monitoring service and says, "I'm alive and watch that I stay alive," periodically passing current state information. In many systems it leases its existence in the Container, and must renew that lease periodically to be considered alive. This is not an alarm, per say. The system assumes that unless state information in the monitor is current, the service is no longer available. When the state is no longer current, the service is regenerated elsewhere. The monitor requests the Service Launcher to deploy the service (again).

Services need a way of finding other services that they may need in their role as components of an application. This is usually called a Discovery service. The service will find the Discovery service (through a constant way such as a known address) and then register itself as an active service. Service monitors and discovery services are similar patterns and sometimes do double duty service. Services usually use a URL as their address and logical name. When a component needs to find a resource service, it requests the location from a Discovery service, and then connects to that remote service URL. Thus applications in self-* systems are a 'logical' construct made up of from a pool of distributed, deployable component/resource services.

So recapping, the 'application' pattern is loaded as metadata into a Registry. A Service Launcher reads this information and automatically deploys the service into a Container on a server in the grid pool. The service gets its start up state from a proxy to the Registry. The service then registers itself with the Monitoring service. Then, to do the job of the application, it goes to the Discovery service to find address of all the component/resources it needs, then it links to them and the application is functioning. Periodically the service renews its healthy state data in the Monitor. If the Monitor finds the healthy state data is expired, it assumes the service is lost, de-registers it from the Discovery service, and then requests the Service Launcher to re-launch the service elsewhere in the logical domain.

This treatment of self-* architecture is short and may vary slightly from self-* system to system. Mostly services today are SOA services, either Java app services,

Jini services, or web services. Component services can be large or small depending on the capacity of the pool servers and the service loader. Services can be short lived or very long lived (sometimes called 'sticky' services.). A single Boot-strap service can launch from scratch, every needed service into a clean grid pool – usually in seconds.

Self-* Networks

In this world, devices should behave much like services/applications. The device will have an active pattern (not as is current, a passive relationship) with the larger system. Devices will support some communication system, usually a web service interface or a java interface. On startup, a device will find the discovery service and register its state. It is required to periodically renew this information and refresh its lease on life. If the monitor finds the state information old, it assumes the device is not functioning. It puts in an automatic service recovery request, often attempting to restart the device before opening a trouble ticket. Note the device leads the health-conversation - not, as is done today, the management system finding the device and driving communications to its MIB. The self-* system takes over when the device is not healthy.

Communications connections are also monitored as a 'logical' service. These are registered as communications links or paths and the device acts as a proxy to maintain the health data in the monitor. If the health data is stale, it is assumed the connection is down and must be attended to. The registering of device state, connections, and their state can be done in a specialized monitor application which then becomes a 'model' of the network. This model can be replicated for specific tasks, such as network capacity simulations. If a two-way action pattern supporting transactions is established between this network model service and the devices, then the model becomes a provisioning service component. The provisioning service places the new connection in the network model, and the network model replicates it to the devices in the network. Note, the network devices understand and implement transactions.

Most frameworks or Service fabrics contain a large set of 'mandatory' component services. These provide common requirements. Security services are an example. Some self-* implement the entire AAA framework. Another component service is a transaction controller; often, many copies of which are deployed as resource services. The most complete frameworks include a collaboration service and other patterns for service orchestration, which support the notion of a service SLA.

Generally, all these orchestrations and interactions are governed by Policy statements. The services act as active 'agents' to enforce this policy. Sometimes there is a work manager that will ensure a flow of data or control occurs according to the goals and needs of the 'logical' application. The Policy service can implement a pattern of activity derived from biology, such as Stigmergy.

Qualitative or discrete stigmergy is the use of different stimuli to trigger different responses. Franziska Klügl explains: "The main problem with the application of this form of co-ordination in artificial systems is its development: one agent has to modify its environment so that the others respond to stimuli in the intended way." The question remains, in a software system, what is the environment? Today, the environment is the self-* framework in which the agents or services interact.

Again, Richard Nicholson explains:

> The signature of any real next generation distributed runtime is easily identifiable. A next generation "Service Fabric" will dynamically instantiate the most sophisticated composite applications from their structural descriptions; scaling, throughput and availability considerations expressed as runtime SLA's associated with each business service. Self-repair, self-scaling, and self-protecting, a next generation "Service Fabric" will have no single points of failure, and will adapt in response to the volatility it experiences in the underlying resource landscape.
>
> A "Service Fabric," rather than forcing business services into a specific execution pattern (GRID, ESB, etc), will instead adapt in a manner appropriate to support the requirements of each instantiated business service. Within a next generation "Service Fabric," initial deployment, service recovery, service adaptation and manageability considerations coalesce, and so can be seen for what they really are; simply different facets of the same fundamental management task, at each point in time ensuring that each business system achieves "target state."

Alarm Processing, a Self-* example approach

With all the existing infrastructure which is already deployed on the paradigm of broadcasting alarms, the current fault applications are not going out of business anytime soon. But service providers could start to draw down on the problem by demanding intelligent devices with active programmatic interfaces from their vendors. By deploying switches/routers which find networks, register themselves, automatically download software updates, get configurations from directories or XML servers, and then push their management interface into the management infrastructure, providers would be on the way to self-healing networks. Devices can be made to do this; two vendors did this in the TMF Fine Grain NGOSS Catalyst demonstration.

Alarm processors and other OSS could be rewritten to be self* but more advanced models can provide a generational jump. As described above, devices are modeled in a virtual/logical representation of a system which connects to the devices, which supply constant state updates. This network model then becomes an active control interface and a ‘picture’ of the health of the network. This model is a set of distributed services within the self-* grid. As the self-* services maintain their own health, these services maintain the health of the network. By replicating services in the grid, standby, regenerative OSS is created - as well as policy-driven scaling to meet real-time demands. By placing an application pattern or a connection pattern into the meta-data of the Registry or Network model, the real instance is created and automatically maintained by the system. It stays up until either there are no more healthy resources available anywhere, or you request it to be ripped down.

Companies and Open Source Libraries

There are several open source movements that target self-* systems. Notably these include the Java community projects for Jini and RIO. Perhaps the largest organization is the Global Grid Forum which produces not just interoperability “standards” but also supports an open source code library. Internet II and OASIS

have strong overlaps with these aforementioned groups.

Most of the first generation self-* startup companies and internal corporate initiatives have already failed. Sun's Jini, MCI's NewWave, Valaran, and IntaMission all broke technical ground, but were unable to turn this into market success. Fortunately, industry progress still occurs, these approaches just will not die. In these articles we try to not endorse any specific vendor solution as better than another; rather to list places the interested reader can go for follow up and solutions - where you can go to get the needed tools, platform, and equipment. Today, 2nd generation self-* products exist and are available for commercial deployment. All these companies have successful deployments past them. Most have endorsed the Grid model and ubiquitous computing as frameworks for their marketing message. While probably not the complete list, these are companies with whom the author has experience:

- Blitz (modified model of supported open source Javaspace): out of England
- Gigaspaces: moved from Israel to the US; also a principle supporter of the open source RIO project
- Majitek: out of Melbourne, Australia
- Netcaboddle: out of England.
- Paremus: out of England
- Univa Corporation (supporting GGF open source): out of the US

In addition, among established, mainstream vendors, IBM has a strong expertise in business support grids and also T-spaces, a space-based product; however, you must be really persistent with the IBM account teams to get them to propose these products. CapGemini Europe has an "integration" sub-practice that is based on self-* software, but again it is hard to extract from their mainstream practice. HP and Sun have long been associated with the concept of agile systems; but neither offer specific self-* products as part of their mainstream offering. Many Japanese companies have active research and projects deploying this technology internally, but have not made any strong commercial push as self-* software or hardware vendors. We know Motorola, current home to policy pioneer John Strassner, also has a program, but it is buried very deep for the moment and we do not know when it will see the light of day.

Engaging the full power of Complex Systems Dynamics

Dr. Daniele Miorandi of CREATE-NET & BIONETS writes:

> We make "a case to go for biology as the "source of inspiration" for breaking such complexity ceiling. The basic idea is to move from the conventional top-down engineering design approach (from use-cases to specifications, then to architecture, implementation and testing, all based on the "divide et impera" paradigm) to a bottom-up approach, in which the "intelligence" lies in the design of an artificial ecosystem, embedding the principle underpinning evolution in simple components, the expected behavior arising from a complex web of interactions." [Proc. of IEEE SMC (http://www.smc2007.org/)]

Further he states,

> "Concerning self-healing networks, my opinion is that - again - nature represents one of the most promising directions to look at. The basic idea shall be here to design reactive systems, where the reaction patterns is not defined at the system design phase, but shall be able to adapt and evolve (again, autonomically) over time."

Fundamentally, study of complex communications has two different solution approaches. For example, CASCADAS, seeking the maximum level of autonomous behavior, postulates that each component unit must embody within itself the self* characteristics. They seek systems with autonomous components. The Fine Grain approach, shared by developing Grid standards and open source projects like RIO, is to embed the self-* characteristics in the matrix of the system interactions. No component is really autonomous; instead the system as a whole is autonomous, as the structures and DNA of the services enforce collaboration to achieve practical self-* applications. This style, we know how to do today.

The futuristic, fully autonomous component is still beyond practical application, but this is no reason to wait for self-adaptive, semantic, policy-driven systems. The first Java application servers were barely able to run a single service; systems improve over time because bright developers and architects see better approaches. Different projects and companies cross fertilize each other. Operations groups identify issues and flag them for priority solution, thereby focusing resources toward generating new behaviors for the whole system. The evolution of software is itself a complex system – one that is adaptive, but it takes leadership to turn that adaptation toward deploying self-* systems.

Leadership for Survival

Rudy Puryear pegs the central issue: "IT reflects the complexity that's been built into business. Unnecessary complexity in the business has resulted in a lot of unnecessary complexity in IT. That's slowing down the cycle times in IT. The cycle times in IT are increasingly much slower and out of sync with the cycle times required by the business to stay competitive." [CIO]

However, there is good complexity and bad complexity – based on how an organization harnesses its complex systems. If you are only reactive, you are bound into system eddies, facing the same problems over and over, or worse you become victim to Darwinian-like negative selection. If you look again at the world and your place in the market as an "ordered system" reflecting complexity, you can begin to develop strategies for success. Every time a new network paradigm or a new class of service product is created, this becomes an opportunity to explore and deploy self-healing systems. To date, our telecommunications ecosystem has not made this commitment to be truly healing. Supporting self-* companies by buying and deploying their products is a step toward reducing costs and controlling the ship-of-*market*-state. Then networks and support systems will make an evolutionary leap to a new plateau.

If you have news you'd like to share with Pipeline, contact us at editor@pipelinepub.com.