Architectural Styles for Runtime Software Adaptation

Richard N. Taylor
University of California, Irvine
Institute for Software Research
Irvine, CA 92697-3455
taylor@ics.uci.edu

Nenad Medvidovic
University of Southern California
Computer Science Department
Los Angeles, CA 90089-0781
neno@usc.edu

Peyman Oreizy
Launch21
peymano@launch21.com

Abstract

Runtime software adaptability – the ability to change an application's behavior during runtime – is an increasingly important capability for systems, both to support continuous operation and to support a good user experience. Achieving such adaptability may be very hard or easy; the degree of difficulty will largely reflect choices made in a system's architecture. Some architectural styles are much more supportive of dynamic change than others. This paper examines a range of styles and assesses them with respect to a four-element evaluation framework, called BASE. The framework considers how a style supports changes to behavior, state, its execution context, and supports asynchrony of change. Styles considered include REST, event-based, service-oriented, and peer-to-peer.

1. Introduction

In an invited paper presented at ICSE 2008 [20], the authors presented a framework for the evaluation and comparison of different techniques for achieving runtime evolution of software systems. The paper also discussed several architectural styles, comparing and contrasting them in terms of the evaluation framework. Because of the required content of the invited paper (a retrospective on our "most influential paper of ICSE 1998" [19]), the consideration of the architectural styles was brief. The objective of this paper is to revisit the styles considered in [20], more carefully evaluating them with regard to the framework, and to expand the discussion to include additional styles. Notably, this paper adds discussion of service-oriented architectures (SOAs) and peer-to-peer styles.

We begin this paper with a very brief introduction to runtime software evolution and our evaluation framework, called BASE, first introduced in [20]. We then consider, in more detail, the styles discussed in the previous paper and then turn our attention to SOAs and to P2P systems. The paper concludes with a call for the careful consideration and development of novel styles, certainly for supporting evolution, but more generally, in ways that capture the rich experience base of the software engineering community.

2. Runtime Software Evolution

By *runtime software evolution* (RSE), or alternatively, *dynamic adaptation*, we refer to the ability of a software system's functionality to be changed during runtime, *without* requiring a system reload or restart. Such systems are increasingly of interest because of the demand for non-stop systems, or even simply to avoid annoying their users.

Numerous techniques have been devised over the years to support RSE. Arguably, the job of an operating system is to support RSE, enabling new programs to be executed without requiring a system reboot or restart. Ironically, operating system "updates" and patches for security vulnerabilities require a system reboot. More interestingly, however, RSE is focused on supporting dynamic change to applications at the user level. For instance, the ability to "upgrade" a Web browser with new plug-ins without requiring browser restart (and hence without requiring the user to re-navigate to his current context) is an example of RSE attuned to the well-being of the user.

3. BASE Framework

Our framework, introduced in [20], provides a means of evaluating, comparing, and combining techniques for runtime evolution. It differentiates techniques based upon the system *model* they operate on and how they confront four aspects of runtime change: behavior, asynchrony, state, and execution context; as a result, we refer to the framework as BASE.

Typical system *models* used for RSE range from high-level abstractions, such as a software architectural model [22], to executable models, such as source code and machine code, to data-flow models, and others. The model determines the *entities* that can be changed (e.g., in an architectural model, the components, con-

nectors, and configuration) and the *operators* of change (e.g., in an architectural model, the addition, removal, and replacement of components and connectors and their interconnections).

When a system model is changed during runtime, four aspects of that runtime change must be addressed:

1. *Behavior:* This concerns how the behavioral specification of the system is changed. What aspects of the behavior can and cannot be changed? Can new behavior be added, can existing behaviors be replaced, or are we restricted to recombining existing behaviors? How are behavioral changes represented, deployed, and instigated? What assurances, if any, are provided that the behavioral changes have not been tampered with?
2. *Asynchrony:* Applying a change during runtime is not instantaneous. It can take anywhere from a few milliseconds to apply a patch to a small program's machine code to minutes or hours to change a large distributed system (e.g., due to communication latency and nodes being unavailable). Is system execution suspended during change or does it continue to run in some capacity? If the latter, how does the approach deal with partially applied changes?
3. *State*: This concerns how the system's state is changed, whether in memory, on disk, or in a separate subsystem, such as a database. How are state changes described? If the definition of a type changes, are existing instances altered? Are all instances updated simultaneously or lazily as they are accessed? Is the execution of the system suspended while state changes are made?
4. *Execution context:* A model is interpreted by some machine, e.g., an 80x86 microprocessor, a Java virtual machine (JVM), a type inference engine, or a data flow engine. As we make runtime changes to our model, we must be careful to not adversely affect the machine that interprets it. For example, we may want to reorder statements in a particular procedure, but the machine executing our program may not support it in situations where the procedure is on the program stack or if the procedure has been inlined within other procedures.

The original paper also discussed *implementation probes* as part of the evaluation framework. While probes are often indispensable for ascertaining whether, e.g., a particular change can be effected, we do not consider this topic to be intrinsically a matter for evaluating RSE techniques. Consequently, we do not focus on this aspect in the remainder of this paper.

This framework is used in the ensuing discussion, and especially in the table at the end of the paper, to help tease out the differences in the various styles considered. Not all styles or adaptation techniques discussed in the following section include means for addressing all four aspects.

4. Styles That Make Adaptation Easier

Making life easier for oneself through use of architectural styles is a long-established practice of developers. Architectural styles are named collections "of architectural design decisions that (1) are applicable in a given development context, (2) constrain architectural design decisions that are specific to a particular system within that context, and (3) elicit beneficial qualities in each resulting system" [27]. Here, the beneficial quality sought is dynamic adaptability.

What makes a style amenable to runtime change? In this section, we describe several styles that support runtime evolution and highlight a few styles that show great promise for the future. For each style, we use the BASE framework to categorize the specific characteristics that facilities dynamic adaptation. A summary of these characteristics is provided in Table 1.

Pipe-and-Filter and Weaves

Style description: The pipe-and-filter style exemplifies explicit, severable components, explicit connectors, the use of common interfaces, and standardization on the root type of all messages (viz., ASCII streams) [24]. Creating a new pipe-and-filter application from existing filters is trivial.

In a pipe-and-filter system each component (filter) is responsible solely for its own processing of the data stream that arrives on its input port and for depositing the processed data on its output port. The pipes route the data streams among the filters. The filters have no dependency on, or knowledge of, their neighboring filters. Moreover, a filter typically does not maintain internal state, meaning that the processing of each chunk of data is entirely self-contained and does not depend on what the filter has done previously. This allows arbitrary addition, removal, replacement, and reconnection of filters (and pipes).

The Weaves system and its accompanying style [12] exploits the strengths of pipe-and-filter and supplies the key missing elements for supporting runtime adaptation. Weaves provides for the dynamic rewiring of pipes through a combination of explicit buffering of messages and flow control mechanisms.

In particular, when a pipe (which is referred to as a "queue" in Weaves, and which is explicitly sized) is detached from a consuming filter (referred to as a "tool fragment" in Weaves), the buffer within the pipe accommodates incoming messages up to its limit. As the buffer's limit is approached, the pipe invokes a standard interface on the producing filter to retard or in-

hibit the production of more messages until the rewiring is complete and the buffer regains capacity for handling additional messages. This suggests that the producing filter may have to maintain internal state during dynamic adaptation, but still, no other filters have any dependencies on that state.

BASE Framework:

- **Behavior:** Tool fragments may be added, removed, and rewired. Weaves can dynamically load the code modules of tool fragments using facilities provided by the operating system.
- **Asynchrony:** Buffering in queues insulates tools from momentary disconnects during rewiring, while flow control supports more disruptive rewiring. Weaves also supports raising/lowering the thread priority of tool fragments to regulate the output and input flows of communicating tool fragments.
- **State:** In the primary application domain of Weaves (satellite telemetry processing), many tool fragments are stateless. Tool fragments that maintain state are designed to be resilient to data flow interruptions.
- **Execution context:** Each tool fragment executes in its own thread of control, which alleviates control flow dependencies among tools. Weaves guarantees the atomicity of sending and receiving data. Since shared memory serves as the primary mechanism for message passing and multiple tool fragments are allowed to simultaneously reference the same message object (yielding performance advantages), message objects must handle synchronization and conflict resolution to guard against data corruption.

Event notifications: Field and Pub-Sub

Style description: In event-based architectures, components do not directly invoke one another, but rather issue notifications of events they have observed or generated during their processing. Those events, in turn, may be of interest to other components, which then perform some operations and possibly issue other events in response. Event-based architectures allow, in principle, arbitrary addition and removal of components since the components are completely decoupled and their interactions are facilitated by an explicit mechanism, usually reified as some sort of event bus. For example, adding a new component will result in possibly new events being generated – which may or may not be of interest to existing components in the system – and existing events (implicitly) invoking the new component's functionality.

An early example of event-based architecture underlies the Field software development environment [23]. Field was created with the express intent of supporting dynamism: independent components (Unix tools) were invoked in response to receipt of explicit events routed through a centralized message connector. The set of tools that would respond to a particular event were not pre-specified and tools could be added to and deleted from the set dynamically.

Efficiencies in the way that Field components communicated could be introduced through declaration of the types of events a component would wish to hear about. Additionally, event generating components would advertise their events, so that event routing tables could be formed, possibly dynamically, implicitly connecting event generators and consumers. This mechanism is conceptually that of publish-subscribe. Numerous pub-sub mechanisms have appeared over the past fifteen years [9] (a notable example is Siena [4]), enabling this basic technique to be effectively applied in a wide variety of circumstances [15].

BASE Framework:

- **Behavior:** Components may be added and removed, possibly resulting in new events being generated.
- **Asynchrony:** Typically, components do not depend on the presence of other components and operate reactively to received events.
- **State:** No explicit support. New components typically probe the system's state to construct needed component state and are expected to not adversely affect the global state of the system if they are removed during runtime.
- **Execution context:** Conceptually, components execute independently of one another wherein each component waits for an event, processes that event, and, optionally, generates new events in response. In practice, components may share memory and a single thread of control, thereby introducing control and data inter-dependencies.

Event-based components and connectors: C2

Style description: C2 is a style that imposes a particular topology on components that otherwise communicate via events [26]. All communication is mediated by active, first-class connectors. A component is attached to at most a single connector on its top and another on its bottom. Two types of events are exchanged by components: requests, which flow upward, and notifications, which flow downward. A component in a C2-style architecture may have knowledge of the components above it (hence its ability to issue requests for specific services), but has no knowledge of the components below (hence its ability to only issue notifications on its bottom side).

The event-based interaction coupled with the topological constraints and active connectors renders the C2 components readily severable from a given architecture. Thus the unit of behavioral change is the component. C2's layering and visibility rules further promoted adaptability, since they strictly limit the latent or implicit dependencies that arise from knowing which component is going to service a particular event.

Another characteristic of C2 that makes it amenable to dynamic adaptation is the lack of shared state among components: the only allowed way for components to interact is via events. The style, however, makes no particular provision for managing a component's active state when the component is replaced. While such facilities have been built into the C2-specific tool support [16], they are external to the style.

BASE Framework:

- **Behavior:** Components may be added, removed, and rewired. C2 can dynamically load the code modules of new components and connectors using facilities provided by the operating system.
- **Asynchrony:** Topological constraints on component dependence limits the set of components that may be affected by a runtime change. If lower-level components are affected by changes to a higher-level component, connectors that buffer messages and otherwise assist with runtime change can be used.
- **State:** No explicit support, though lack of shared state among components makes the style amenable to state migration strategies. New components typically construct needed state incrementally, from the event traffic.
- **Execution context:** Conceptually, components execute in their own thread of control and without shared state. This obviates control flow dependencies and data synchronization issues. The C2 infrastructure permits multiple components to share the same thread of control, and, in those situations, the infrastructure is careful to ensure that control leaves a component before that component is removed. Likewise, multiple components may execute in the same memory address space, but since data is not shared between components, data synchronization and corruption is not a concern.

Map-Reduce

Style description: The Map-Reduce style [7] parallelizes computations over large data sets by distributing the work across a collection of processing nodes. The desired computation is divided into two phases: a *map* phase, which is given a chunk of the data set as input and outputs a set of key/value paris (k,v), and a *reduce* phase, which is given all values of a particular key (k, (v1...vn)) and outputs the final result of the computation. Given the data set, the map function, and the reduce function, the Map-Reduce infrastructure splits the data set into independent chunks and executes the map phase in parallel among the processing nodes. It tracks the progress of the nodes and, if a node fails, reassigns the work to another node. It then sorts the intermediate key/value pairs and executes the reduce phase in parallel among the processing nodes.

By partitioning its computation and data, a Map-Reduce-based system can efficiently parallelize and load balance a computation across a set of nodes and cope with node failures – i.e. adapting on the fly.

As an example, Yahoo recently used its Map-Reduce infrastructure, called Hadoop, to run the terabyte sort benchmark using over 900 nodes in just under 300 seconds [30].

BASE Framework:

- **Behavior:** The Map-Reduce infrastructure can alter one aspect of system behavior, namely, it can speed up and slow down the rate of computation by adding and removing nodes during runtime.
- **Asynchrony:** Asynchrony is managed centrally by the infrastructure: nodes do not communicate directly with one another, but operate exclusively on the chunks of data provided by the infrastructure. The infrastructure decides if and when to start a node and whether or not to discard its output.
- **State:** Like asynchrony, the impact on system state as runtime changes are made is centrally managed by the infrastructure. When speeding up the computation, unprocessed chunks of the data set are distributed to new nodes. When slowing down the computation or in the event of a node failure, the node's output is discarded and its chunk of the data set is requeued for processing by other nodes.
- **Execution context:** Nodes execute in their own thread of control, without shared state. This obviates control flow dependencies and data synchronization issues. Data flow dependence among nodes – where the output of the map phase is sorted and fed as input to the reduce phase – is indirect and centrally mediated by the Map-Reduce infrastructure. As such, issues such as partial or corrupted data are avoided.

Externalizing State: REST

Style description: The World Wide Web is arguably the world's most successful dynamically adaptable system. The Web is continually changing as clients, servers, proxies, and gateways come and go with staggering rapidity. The application is so dynamic that no complete representation of its architecture "right now" is possible.

The Web owes its dynamic adaptability to its architectural style; indeed, the dynamism is so extreme that the Web is only understandable and describable in terms of its architectural style, which is known as REpresentational State Transfer, or REST [10]. REST has been the principal style of the Web since the mid-1990's, guiding the design of the HTTP/1.1 protocol and the other key elements of the Web. (N.B. there are elements of the Web now which do *not* hew to the REST style; some of these deviations are considered below in the subsection on CREST.)

Designed to support the network exchange of hypermedia documents while preserving component inde-

pendence (thus providing integration scaling in the face of agency borders) and minimizing latency (providing one aspect of performance scaling), REST can be summarized as six simple principles [27]:

1. The key abstraction of information is a resource, named by an URL
2. The representation of a resource is a sequence of bytes plus metadata to describe those bytes
3. All interactions are context-free
4. Only a few primitive operations are available
5. Idempotent operations and representation meta-data are encouraged in support of caching
6. The presence of intermediaries is promoted

Each of the six principles contributes to the scaling, diversity, and ease of adaptation seen within REST's most notable use, the Web. URLs are an anarchic, decoupled namespace with no central authority. Each Web server may support whatever URLs it chooses and assign whatever meaning it deems appropriate to each URL. The freedom to introduce server-specific namespaces and resources is partially responsible for the outpouring of innovative Web sites and uses.

In response to receipt of a request, a Web server transmits a *representation* of the resource requested. The type of the resource is not constrained by the style; rather, interoperability is achieved through reference to an open standard (MIME) for meta-data naming the type of the representation transmitted. Hence the types of resources maintained by servers may change without constraint; the types of representations sent may also change, albeit more slowly, through extension of MIME types. Browsers may be changed independently to display newly developed MIME types; well-designed browsers will "gracefully" handle MIME types with which they are not familiar.

REST principle #3, "all interactions are context-free" is perhaps the most critical in yielding the Web's scalability – a particular kind of adaptability. Sometimes phrased rather misleadingly as "the protocol is stateless", the principle demands that all state be externalized. A request sent to a server must carry with it whatever state is necessary for the server to process the request, without requiring recourse to any prior history of interaction. This is the property that enables the scaling of server farms. Repeated interactions between, e.g., an end user and a book retailer need not always transpire through the same server. Rather, any of the retailer's servers – including newly activated ones – will be able to service the requests. Similarly, a response message to a client browser can provide complete hypertextual context, determining the set of navigation options for the user.

Principle #4 keeps the barrier for introducing new processing components low, as the number of operations that must be supported is small (in HTTP/1.1 these include GET, POST, PUT, and DELETE). Principles 5 and 6 also support adaptability, as support for caching promotes dynamic optimization of request servicing.

The preceding description of REST and its application in the Web hints at how it exploits some key adaptation leverage points. In the case of the Web, clients, servers, and proxies, for instance, are explicit, identifiable run-time elements whose interactions (bindings) are orchestrated by the HTTP/1.1 protocol. This enables their ready manipulation. Explicit message exchange using plain text-based meta-data enables intermediaries to manage interactions. Context-free interactions enable the management of state, for state is *externalized.*

BASE Framework:

- **Behavior:** Web servers and browsers may be added and removed arbitrarily. Both servers and browsers may support whatever set of MIME types they choose, and may change that set arbitrarily.
- **Asynchrony:** Since all interaction are context-free, there are no dependencies among the components comprising the Web at any given moment. Intermediaries, namely caches, enable servicing of requests while a server is unavailable.
- **State:** Since all state is externalized in the HTTP requests, the REST style does not need to provide separate facilities to manage state during adaptation.
- **Execution context:** All Web servers, browsers, and intermediaries execute in their own threads of control. This obviates control flow dependencies and data synchronization issues.

Computational REST

Style description: The preceding discussion of REST mentioned that not all behaviors seen in today's Web are consistent with the REST architectural style. One example is the increasing presence of computation on clients, such as performed by JavaScript or as seen in AJAX applications. Erenkrantz et al. have studied these variations in detail and synthesized a new architectural style, Computational REST [8], that is notable in (at least) the ways it supports dynamism.

Computational REST, or CREST, can be understood as a generalization of REST in which *computations* are the unit of exchange, rather than REST's *representations*. That is, in REST a server returns a static representation of a resource – frequently as HTML – to a browser for display. In CREST, servers can send computations to a client for further execution. Computations may be, e.g., transformation of an image, a temporal service (such as "tomorrow's projected cloud cover over LAX"), a generated collection of other resources, or a simulation of an object – such as of a building's power consumption.

More precisely, erasing the distinction between client and server, CREST peers exchange computations for evaluation. Each such ongoing evaluation may communicate (via asynchronous message passing) with other ongoing CREST execution loci, known by their URLs. Computations are *instructions* plus *state* plus *meta-narrative*. Instructions are a sequence of commands in the instruction set of a URL's interpreter/ virtual machine. State is a virtual machine state, including memory, providing (at least partial) context for instruction evaluation. The meta-narrative is either static metadata describing significant elements and aspects of the computation or reflectively, is itself a computation.

While the details of CREST are complex, and space constraints prevent their full presentation, the upshot regarding dynamism is clear. While REST externalized state in communication and hence promoted a variety of kinds of adaptation, CREST externalizes state, commands that are to operate on the state, and how a collection of commands are to be orchestrated to accomplish some larger goal. By externalizing these additional elements greater control is afforded over changing behavior as well as managing the execution context aspect of runtime adaptation. Thus, for instance, achieving application quiescence may be dramatically eased, for the meta-narrative overseeing a computation may simply halt a computation, send it "away", change the execution locus (e.g. to improve some locally offered service), retrieve the computation, and resume it on the newly improved peer. Moreover REST makes a fairly sharp distinction between clients and servers, wherein computation is confined to servers; clients (browsers) just present data. CREST erases this distinction, enabling adaptation throughout a system. CREST thus affords the prospect of bringing the degree of adaptivity seen in the current Web to all manner of multi-party distributed computation.

Mobile agent-based architectures [1, 5, 25, 29] act as representatives of a particular party in moving from computer to computer to collect and share relevant information. These architectures are related to CREST and, to the extent they externalize computations and state, offer similar benefits with respect to adaptation.

BASE Framework:

- **Behavior:** CREST peers may join and leave arbitrarily. Computations are exchanged among the peers, allowing for arbitrary behavior to be achieved at runtime in principle.
- **Asynchrony:** Since all interaction are context-free, there are no dependencies among the CREST peers.
- **State:** All aspects of state are made explicit and externalized via CREST computations.
- **Execution context:** All CREST peers execute on separate virtual machines. The particular execution context of a given computation is entirely encapsulated within that computation.

Service-Oriented Architectures

Style description: For the purposes of this discussion we consider service-oriented architectures to comprise multiple functions ("services") existing in their own name and address spaces, interacting remotely over a network using some protocol [21]. In most respects SOAs are simply distributed applications. Ostensibly, SOAs are distinguished from other distributed systems because:

- the services may be under separate authorities;
- the services may be implemented in different languages and execute on a variety of machines; and
- services may be dynamically discovered and composed to perform some business activity.

The presence of multiple, independent authorities managing a network of services introduces new needs: the need to manage trust and application security across authority boundaries, and the need to manage the possibility that services may be changed by providers without adequate notification to the users. That services can change unexpectedly is the most relevant for discussion of runtime adaptation: one common thread in SOA development has been the introduction of different decoupling mechanisms intended to (at least partially) shield one service from changes to another, or to allow the dynamic discovery and replacement of one service by another.

Decoupling mechanisms are seen in CORBA [28] or CORBA-like mechanisms in the early years of SOAs, SOAP-over-HTTP in the Web Services world, and more recently "enterprise software buses," which are a form of event notification/message-oriented middleware with application orchestration achieved through use of workflow languages. A mechanism common to all these technologies is the use of delayed binding between the name of the service requested and the actual service provider. In CORBA, this indirection is done through a naming or trading service. In SOAP-over-HTTP, the service requested is described in an XML document encased in a SOAP envelope [3, 17]. Upon receipt (by whatever authority receives the HTTP message) the agent opens the envelope, parses the XML, and dynamically determines what local service to invoke. Parameters to the service are also passed in the SOAP envelope. The format for the contents of a SOAP envelope, however, is application-specific, which limits how intermediaries can interpret, cache, or redirect requests or results. In enterprise service buses, message routing can be simple (e.g., direct addressing, static routing) or complex (e.g., rules-, policy-, or content-based routing).

Change Aspect / Arch. Style	***Behavior***	***Asynchrony of change***	***State***	***Execution context***
Pipe and Filter/ Weaves	Changes to dataflow model mapped to implementation; can add/remove components and connectors; can change flow of data	Flow control in connectors (buffering, rate mgmt, etc.)	No state sharing among components; any management of internal components state during dynamic adaptation is outside the style's purview	No explicit support in pipe-and-filter; Weaves allows data buffering and suspension of data production during dynamic adaptation
Event Notifications	Publishers and subscribers can join and depart	Varies; pub-sub bus can buffer events, etc.	No state sharing among components; any management of internal components state during dynamic adaptation is outside the style's purview	Intentional lack of explicit support in the style; considered an application-level concern
C2	Changes to architectural model mapped to implementation; can add/remove components and connectors; can change topology	All communication is asynchronous, so can exploit control in connectors	No state sharing among components; any management of internal components state during dynamic adaptation is outside the style's purview	Various techniques resting on explicit connectors (e.g., multi-versioning connectors)
Map/Reduce	Infrastructure can speed-up and slow-down the computation by adding and removing nodes; if a node fails, its chunk of the data set can be re-queued for another node	Nodes are independent; infrastructure distributes work to nodes	Infrastructure splits data set and assigns to nodes; if a node fails, infrastructure can discard its output and re-queue the work	
REST	HTTP servers – which are stateless – can be restarted for updates. Browsers may be updated independently by users	Various techniques, e.g., direct all requests to ½ nodes, update idle nodes, direct all requests to updated nodes, update remaining nodes. Receipt of partial messages should be handled gracefully.	State is externalized: all messages carry state and may be inspected; HTTP server is stateless. State stored externally, in, e.g., a database, and updated separately	Before update of a server, e.g., "drain" in-process requests and refuse new requests.
CREST	Stateless servers (peers) may offer URL-specific interpreters; nominal behavior is encapsulated in computations that are transmitted	Same as REST	All aspects of a computation's state made explicit and externalized	Fully included within the computations (i.e., continuations) exchanged between peers
Service-Oriented Architectures	Service providers may autonomously change functions	A service provider may pause accepting new requests until existing calls have been completed and the update made. No coordination across service providers.	Protocols are not required to be context free; no explicit support for state update provided	Caller and callee are in separate address spaces on separate machines (nominally)
Peer-to-Peer	Entire peers are replaced as the unit of change. Peer addition/removal supported as key aspect of the P2P protocol.	Failure of peers is an expected phenomenon, hence peers can be replaced at any time	State is typically replicated across peers and updated frequently	Peers are application programs, replaced wholesale.

While SOAs leverage decoupling and dynamic binding between callers and callees as a mechanism to enable dynamic evolution, coordination of evolution across services is not strongly addressed in the SOA community. While there is intense focus on, e.g., routing and orchestrating individual messages, there has been little attention paid thus far to orchestrating changes across services and authority boundaries.

BASE Framework:

- **Behavior:** Service providers are completely independent of one another and can change the behavior of their services.
- **Asynchrony:** There is no coordination across service providers. Individual providers may implement different policies for receiving and handling multiple requests simultaneously.
- **State:** While individual SOA implementations may provide custom state management support during dynamic adaptation, no such support is provided by the style itself.
- **Execution context:** The service providers and clients execute on separate machines. There is no assumption of a homogeneous execution context.

Peer-to-Peer Architectures

Style description: The peer-to-peer style is characterized by a distributed collection of nodes (called peers) that communicate using an overlay network [18]. Generally, all peers provide identical services, though some P2P systems introduce special peers for bootstrapping purposes or centralizing particular services. Peers are often independent and can join and leave the network on an ad hoc basis. In Skype (www.skype.com), for example, each peer is an instance of the Skype application running on a user's machine, and millions of peers form a world-wide network to support audio and video conferencing amongst users [2]. In the earlier Gnutella P2P system, each peer is an application running on a user's machine that implements the Gnutella protocol [13]. The protocol was focused on search and file sharing. BitTorrent [6] is similar, with each peer focused on efficiently acquiring a copy of a large file from across the network, while simultaneously assisting other peers in performing the same task through uploading chunks of the media file already acquired. Coordination of BitTorrent peers is performed by master nodes.

P2P systems are fundamentally built to support failure or "unexplained absence" of previously active peers, as well as unpredicted addition of arbitrary new peers. System state is distributed amongst all peers and redundantly stored to cope with peers joining and leaving the network. Peers expect other peers to leave unexpectedly and new peers to probe the state of the system as a matter of course. For example, the name and location of a new Skype user joining the network is replicated on multiple peers. As such, discrepancies between peers is possible and must be addressed.

In large P2P networks, it is impractical to assume that all peers can be upgraded in unison. P2P protocols often provide a means for peers to declare or query the capabilities of other peers. Thus, the overall system supports runtime behavioral change one peer at a time: a peer leaves the network, updates its implementation, and rejoins the network. Change is inherently asynchronous in such a system.

BASE Framework:

- **Behavior:** Peers may be added and removed arbitrarily, and may provide arbitrary functionality.
- **Asynchrony:** There is no requirement that the entire P2P system be updated in unison. Each peer may join or leave the system independently of other peers. Dynamic change to P2P systems is hence inherently incremental and asynchronous.
- **State:** System state is distributed among the peers and redundantly stored. New peers are expected to probe the state of the system.
- **Execution context:** All peers execute in their own threads of control, likely within designated sandboxes on their local hosts. There is no assumption of a homogeneous execution context. Furthermore, specially designated master peers may redirect computation to specific nodes, e.g., to balance the computational or communication loads.

5. Discussion

The "adaptable styles" surveyed above exhibit a wide variety of techniques for addressing the key elements of the BASE framework: the needs of changing behavior, updating state, handling the asynchrony inherent in change, and addressing the needs of updating the execution context. These differences are summarized in the table on the preceding page.

What is to be learned from this large set of techniques? (1) They were developed to address particular types of adaptation needs. (2) They were codified in terms of design rules and principles. (3) They were applied during system implementation. In other words, the applications were designed to be adaptable by following design rules –styles– that make that adaptation possible and manageable.

A more substantive insight can be phrased in terms of the adaptation *leverage points* found in the different styles, i.e. common techniques used by these architectural styles to achieve adaptability. We argued in [20] that the central techniques that have been successful in facilitating adaptation – and which are essential in new styles that are designed to support adaptation – are:

- making the parts that are subject to change identifiable and manipulable;

- providing mechanisms for controlling interactions with the parts subject to change; and
- providing techniques for managing state.

Arguably this is just another way of looking at the core elements of BASE. However, reflecting on them in the context of the preceding table has helped to identify two general strategies: make bindings malleable and use explicit events/messages in communication.

Malleable binding means that an entity is not necessarily linked to its usage context until the time during execution when the computation cannot proceed further in absence of a specific linkage; moreover such linkage can be reset again, later, to meet adaptation needs. The key is that this concept can be applied at higher levels in the abstraction hierarchy than execution. As we previously wrote [20], a plug-in component, for instance, may remain unbound to a master application until, e.g. application initialization, or even until the point at which the plug-in would be called to perform some service. Likewise, clients may only be bound to a particular server once a load balancer determines which server out of some class has the best chance of quickly attending to the client. Messages may be delayed in binding to a specific destination, as they are routed across intermediaries. Malleable bindings means that adaptation comes "for free" with respect to the given binding context.

This concept is used in several existing approaches that leverage styles to enable dynamic adaptation. For example, Rainbow [11] couples an architectural style with its corresponding adaptation style and actively maintains the mapping between the architecture and implemented system. Thus, any changes to the architecture are guided by its style and subsequently reified (i.e., bound) to the corresponding changes in the system. Similarly, Kramer and Magee's layered reference architecture for supporting self-adaptive systems [14] binds the architectural decision-making facilities at different layers only as necessary: each layer is expected to handle its own adaptation in principle and only when it is unable to do so does it seek help of the (meta-) layer above.

As we have earlier argued, use of explicit, or "first-class" events/messages in communication facilitates adaptation. This happens in two main ways. First, such communication implies a strong decoupling of the components engaged in the communication: no direct procedure calls are involved, no shared memory, no tight bindings. Second, first-class events are encapsulated entities, so they may be examined, logged, or manipulated by third-parties to serve various adaptation needs. They may have associated meta-data (as occurs in HTTP/1.1 for example) that enable reasoning about what kinds of processing services they may require upon reaching their destination.

Explicit messages are leveraged to varying degrees by existing architecture-based adaptation techniques. For example, C2 [26] and Weaves [12] were incorporated into our early message-driven architectural adaptation framework [19]. This framework has been a key influence for a number of subsequent approaches, including Rainbow as well as a couple of recent techniques for building self-adaptive robotics applications [31, 32].

6. Conclusion

There is little doubt that the need for self-adapting, autonomous systems is going to increase. It is almost a corollary that much of such adaptation will happen dynamically, while the application remains in at least some degree of continuous execution. Designing such applications demands thought beforehand regarding the technique(s) to use to enable the desired kind of RSE. In some cases the most effective strategy will be to simply see which of the many existing strategies, such as those discussed above, match the adaptation needs of the new application, and follow the best one.

More challenging, and more interesting from the research perspective, is tackling those situations where existing strategies do not suffice. Such needs may arise from, say, performance demands, or complicated state and transaction management needs. Whatever the reason, a thoughtful analysis may yield a novel architectural style, applicable not just to the immediate task, but more generally. The peer-to-peer style discussed earlier arose from the desire to "share" digital media — yet the resulting style has proven to be of value in a wide variety of circumstances, including the robust telephony provided by Skype.

We believe that as a community we should be actively promoting the development of new architectural styles and concomitant adaptation techniques. Such contributions are constructive, helping us all in the development of new systems.

7. Acknowledgments

The work was sponsored by the National Science Foundation under Grant numbers ITR-0312780, SRS-0820222, and SRS-0820170. The authors wish to thank Eric Dashofy for his many useful insights that have made their way into this paper.

8. References

[1] Agha, G. and Jamali, N. CyberOrgs: A Model for Decentralized Resource Control in Multi Agent Systems. In Proceedings of the Workshop on Representations and Approaches for Time-Critical Decentralized Resource/Role/Task Allocation, Melbourne, Australia, July 2003.

[2] Baset, S.A. and Schulzrinne, H. An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol.

Columbia University, Report CUCS-039-04, 2004.

[3] Box, D., et al. Simple Object Access Protocol (SOAP) 1.1. <http://www.w3.org/TR/SOAP/>, World Wide Web Consortium (W3C), 2000.

[4] Carzaniga, A., et al. Design and Evaluation of a Wide-Area Event Notification Service. ACM Trans. on Computer Systems. 9(3), August, 2001.

[5] Cockayne, W. and Zyda, M. Mobile Agents. Manning: Greenwich, 1998.

[6] Cohen, B. Incentives Build Robustness in BitTorrent. <http://www.bittorrent.org/bittorrentecon.pdf>, BitTorrent.org, 2003.

[7] Dean, J. and Ghernawat, S. MapReduce: Simplified Data Processing on Large Clusters In Proceedings of the OSDI'04: Sixth Symposium on Operating System Design and Implementation. San Francisco, CA, December, 2004..

[8] Erenkrantz, J.R., et al. From Representations to Computations: The Evolution of Web Architectures. In Proceedings of ESEC/FSE 2007, p. 255-264, Dubrovnik, Croatia, Sept 3-7, 2007.

[9] Eugster, P.T., Felber, P.A., et al. The many faces of publish/subscribe. ACM Computing Surveys. 35(2), p. 114-131, 2003.

[10] Fielding, R.T. and Taylor, R.N. Principled Design of the Modern Web Architecture. ACM Transactions on Internet Technology, 2(2), p. 115-150, May, 2002.

[11] Garlan, D., et al. Rainbow: Architecture-based self-adaptation with reusable infrastructure. IEEE Computer. 37(10), p. 46-54, October, 2004.

[12] Gorlick, M.M. and Razouk, R.R. Using Weaves for Software Construction and Analysis. In Proceedings of ICSE'91. p. 23-34, May, 1991.

[13] Kan, G. Gnutella. In Peer-to-Peer: Harnessing the Power of Disruptive Technologies, Oram, A. ed. p. 94-122, O'Reilly, 2001.

[14] Kramer, J. and Magee, J. Self-Managed Systems: An Architectural Challenge In Future of Software Engineering 2007, Briand, L. and Wolf, A. eds. IEEE-CS Press, 2007.

[15] Luckham, D. The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley, 2002.

[16] Medvidovic, N., et al. A Language and Environment for Architecture-Based Software Development and Evolution. In Proceedings of ICSE '99, p. 44-53, Los Angeles, CA, May 1999.

[17] Mitra, N. Simple Object Access Protocol (SOAP) 1.2: Primer. <http://www.w3.org/TR/soap12-part0/>, W3C, 2003.

[18] Oram, A., Minar, N., et al. Peer-to-Peer: Harnessing the Power of Disruptive Technologies. 1st ed. 432 pgs., O'Reilly & Associates, 2001.

[19] Oreizy, P., et al. Architecture-Based Runtime Software Evolution. In Proceedings of ICSE '98, p. 177-186, Kyoto, Japan, April, 1998.

[20] Oreizy, P., et al. Runtime software adaptation: Framework, approaches, and styles. In Companion Proceedings of ICSE 2008, Leipzig, May 2008.

[21] Papazoglou, M.P. Service-Oriented Computing: Concepts, Characteristics and Directions. Fourth International Conference on Web Information Systems Engineering (WISE'03). p. 3, 2003.

[22] Perry, D.E. and Wolf, A.L. Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes. 17(4), October, 1992.

[23] Reiss, S.P. Connecting Tools Using Message Passing in the Field Environment. IEEE Software. 7(4), p. 57-66, July, 1990.

[24] Shaw, M. and Garlan, D. Software Architecture: Perpectives on an Emerging Discipline. Prentice Hall, 1996.

[25] Silva, A., et al. Towards a Reference Model for Surveying Mobile Agent Systems. Autonomous Agents and Multi-Agent Systems, 2001.

[26] Taylor, R.N., et al. A Component- and Message-Based Architectural Style for GUI Software. IEEE Transactions on Software Engineering, June 1996.

[27] Taylor, R.N., Medvidovic, N., et al. Software Architecture: Foundations, Theory, and Practice. 736 pgs., John Wiley & Sons, 2008.

[28] Vinoski, S. CORBA: Integrating diverse applications within distributed heterogeneous environments. IEEE Communications. 14(2), February 1997.

[29] Wong, D., et al. Concordia: An Infrastructure for Collaborating Mobile Agents. In Proceedings of the First International Workshop on Mobile Agents. Berlin, Germany, April 7-8, 1997.

[30] O. O'Malley. Apache Hadoop Winds Terabyte Sort Benchmark. <http://developer.yahoo.net/blogs/hadoop/2008/07/apache_hadoop_wins_terabyte_sort_benchmark.html>

[31] Georgas, J. and Taylor, R.N. Policy-based self-adaptive architectures: a feasibility study in the robotics domain. In Proceedings of SEAMS 2008, pages 105–112, Leipzig, Germany, May 2008.

[32] Edwards, G., et al. Architecture-Driven Self-Adaptation and Self-Management in Robotics Systems. In Proceedings of SEAMS 2009, Vancouver, Canada, May 2009.