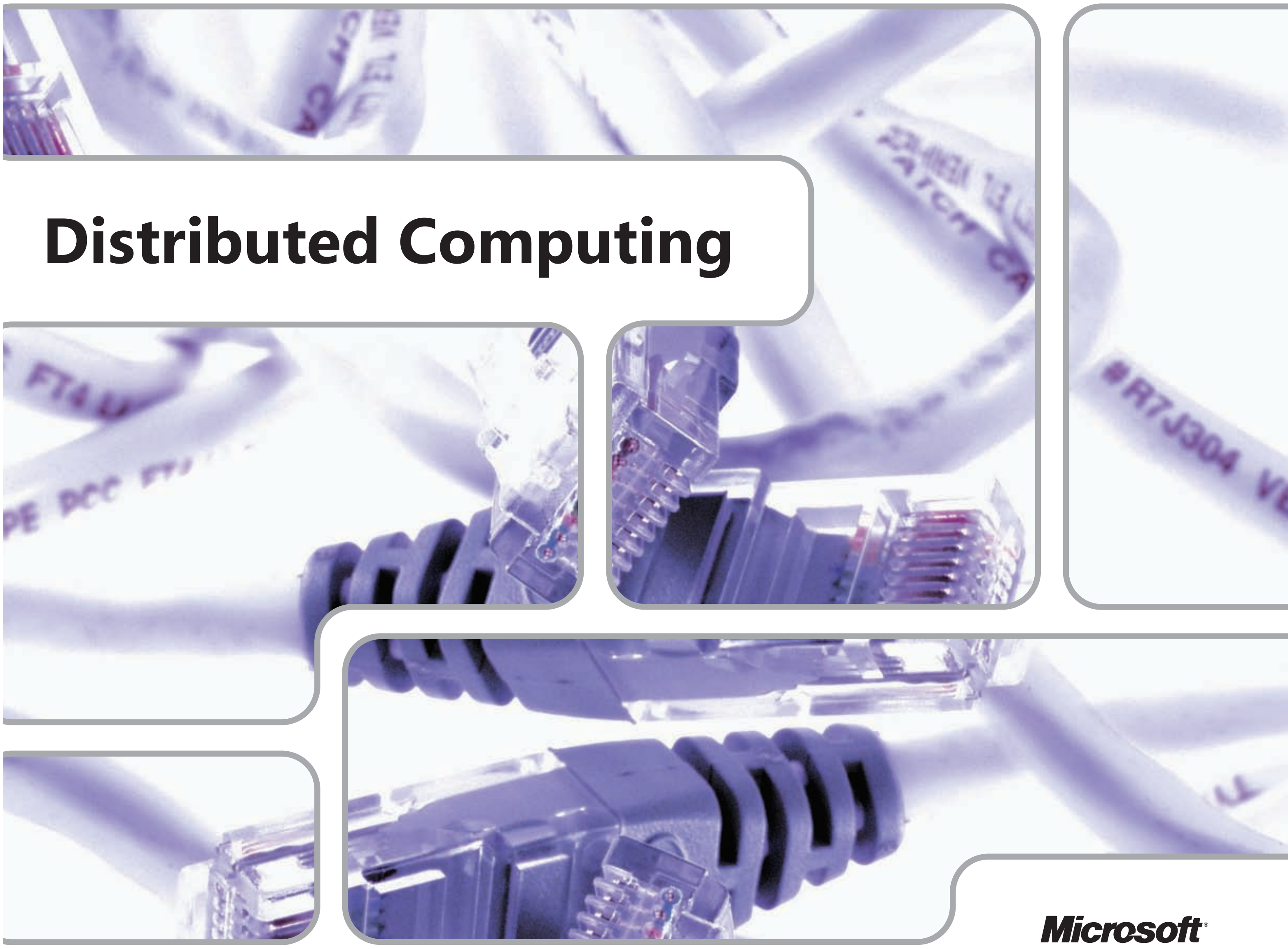
The Architecture Journal

input for better outcomes

Learn the discipline, pursue the art, and contribute ideas at www.architecturejournal.net

17



Distributed Computing

Microsoft

Considerations for Designing Distributed Systems

Architectural Patterns for Distributed Computing

Head in the Cloud, Feet on the Ground

Architecture Journal Profile: Microsoft Robotics Group

On Distributed Embedded Systems

Using Events in Highly Distributed Architectures

Caching in the Distributed Environment

Distributed Applications in Manufacturing Environments

Contents

Foreword 1
by Diego Dagum

Considerations for Designing Distributed Systems 2
by Marc Mercuri
A discussion about the distributed systems momentum, spotlighting its remaining challenges and considerations.

Architectural Patterns for Distributed Computing 6
by Joshy Joseph, Jason Hogg, Dmitri Ossipov, Massimo Mascaro, and Danny Garber
Lessons learned from years of field practice on service design, deployment, consumption, and operation.

Head in the Cloud, Feet on the Ground 14
by Eugenio Pace and Gianpaolo Carraro
Eugenio and Gianpaolo cover a deep analysis of the trade-offs of distributed cloud computing.

Architecture Journal Profile: Microsoft Robotics Group 20
Henrik Frystyk Nielsen and Tandy Trower are working to bring the age of personal robotics to the horizon. Find out what they have to say about their careers and the future of computing.

On Distributed Embedded Systems 22
by Arvindra Sehmi
An exploration of how DES differs from traditional distributed systems, its market opportunity, challenges, and scenarios.

Using Events in Highly Distributed Architectures 29
by David Chou
SOA succeeded in getting loose coupling at a technical level; now let's go for the same at a business-process level.

Caching in the Distributed Environment 34
by Abhijit Gadkari
Responsiveness through caching techniques, a collateral aspect of distributed systems with direct impact on user experience.

Distributed Applications in Manufacturing Environments 40
by Peter Koen and Christian Strömsdörfer
Beyond the limits of memory-based objects, here is an architecture paradigm to deal with concrete, real-world ones.

Founder
Arvindra Sehmi

Director
Simon Guest

Editor-in-Chief
Diego Dagum

Contributors for This Issue
Gianpaolo Carraro
David Chou
Abhijit Gadkari
Danny Garber
Jason Hogg
Joshy Joseph
Peter Koen
Massimo Mascaro
Marc Mercuri
Henrik Frystyk Nielsen
Dmitri Ossipov
Eugenio Pace
Tandy Trower
Arvindra Sehmi
Christian Strömsdörfer

Design, Print, and Distribution
United Business Media Limited – Contract Publishing
Chris Harding, Managing Director
Angela Duarte, Publication Manager
Bob Steigleider, Production Manager
Camille Verde, Senior Art Director

Dear Architect,

In the early days in this industry, making computers interoperate—while not impossible—was so complex that the effort was only expended in very compelling business cases. Back then, concurrent and distributed programming were specialties in computer science—the programming models were obscure and, from the programmer's perspective, utterly focused on cross-platform issues of communication, coordination, and fault tolerance.

How things have changed—*Do you want to target an undetermined, location-unaware universe of users?* Host your application on an Internet accessible Web server. *Do you still need to reach some legacy systems whose investment and maturity don't justify a migration?* Check whether WS-* or REST standard stacks, each with its strengths and weaknesses, are suitable to bridge those existing applications. *Do you need to tackle intermittent connections and high latency times?* Consider deploying a small portion of your application to a desktop—

Today, nanotechnology makes it possible to host a powerful CPU anywhere and the pervasive Internet, together with simple, lightweight programming models, connects it to the world. Isolated single processor computing is a shrinking universe. Embedded systems, robotics, and parallel and distributed computing are becoming the norm and redefining the edges of user experience.

Marc Mercuri starts off the discussion with an overview of the challenges of distributed systems design. Joshy Joseph and colleagues mine years of field practice for architectural patterns for service design, deployment, consumption, and operation. Gianpaolo Carraro and Eugenio Pace explore the concrete trade-offs of Software-as-a-Service (SaaS)-model distributed cloud computing.

We pause along the way for a conversation with Henrik Frystyk Nielsen from the Microsoft Robotics Group, accompanied by Tandy Trower, general manager of the group.

Back on track, Arvindra Sehmi explains how Distributed Embedded Systems differs from traditional distributed systems in terms of market opportunity, challenges and scenarios. Then David Chou covers Event-Driven Architectures (EDA), an architectural style proposed as the next iteration of Service-Oriented Architecture (SOA).

Abhijit Gadkari addresses responsiveness through caching techniques, touching on the Velocity project—a distributed, in-memory application cache platform for developing scalable, available, and high-performance applications.

Finally, Christian Strömsdörfer and Peter Koen go beyond the limits of virtual, memory-based objects proposing an architectural paradigm for dealing with physical environments in the real world.

I sincerely hope you enjoy this issue. You may have noticed the new look of the magazine. This redesign is part of a wider initiative bringing together all our architecture-related materials under one brand. As always, we welcome your feedback at editors@architecturejournal.net.

[illegible]

Diego Dagum

Considerations for Designing Distributed Systems

by Marc Mercuri

Summary

We're at an exciting crossroads for distributed computing. Historically considered something of a black art, distributed computing is now being brought to the masses. Emerging programming models empower mere mortals to develop systems that are not only distributed, but concurrent – allowing for the ability to both scale up to use all the cores in a single machine and out across a farm.

The emergence of these programming models is happening in tandem with the entry of pay-as-you-go cloud offerings for services, compute, and storage. No longer do you need to make significant up-front investments in physical hardware, a specialized environment in which to house it, or a full-time staff who will maintain it. The promise of cloud computing is that anyone with an idea, a credit card, and some coding skills has the opportunity to develop scalable distributed systems.

With the major barriers to entry being lowered, there will without a doubt be a large influx of individuals and corporations that will either introduce new offerings or expand on existing ones – be it with their own first-party services, adding value on top of the services of others in new composite services, or distributed systems that use both categories of services as building blocks.

In the recent past, "distributed systems" in practice were often distributed across a number of nodes within a single Enterprise or a close group of business partners. As we move to this new model, they become Distributed with a capital D. This may be distributed across tenants on the same host, located across multiple external hosts, or a mix of internal and externally hosted services.

One of the much-heralded opportunities in this brave new world is the creation of composite commercial services. Specifically, the ability for an individual or a corporation to add incremental value to third-party services and then resell those compositions is cited as both an accelerator for innovation and a revenue generation opportunity where individuals can participate at various levels.

This is indeed a very powerful opportunity, but looking below the surface, there are a number of considerations for architecting this next generation of systems. This article aims to highlight some of these considerations and initiate discussion in the community.

Reference Scenario #1 – Fabrikam Package Tracking Service

When talking about composite services, it's always easier if you have an example to reference. In this article, we'll refer back to a scenario involving a simple composite service that includes four parties: Duwamish Delivery, Contoso Mobile, Fabrikam, and AdventureWorks (see Figure 1).

Duwamish Delivery is an international company that delivers packages. It offers a metered Web service that allows its consumers to determine the current status of a package in transit.

Contoso Mobile is a mobile phone company that provides a metered service that allows a service consumer to send text messages to mobile phones.

Fabrikam provides a composite service that consumes both the Duwamish Delivery Service and the Contoso Mobile Service. The composite service is a long-running workflow that monitors the status of a package, and the day before the package is due to be delivered, sends a text message notification to the recipient.

AdventureWorks is a retail Web site that sells bicycles and related equipment. Fabrikam's service is consumed by the AdventureWorks ordering system, and is called each time a new order is shipped.

Reference Scenario #2 – Product Finder Service

A delta between supply and demand, whether the product is a gadget, a toy, or a ticket, is not uncommon. Oftentimes, there are multiple vendors of these products with varying supply and price.

A. Datum Corporation sees this as an opportunity for a composite service and will build a product locator service that consumes first-party services from popular product-related service providers such as Amazon, Ebay, Yahoo, CNET, and others (see Figure 2).

The service allows consumers to specify a product name or UPC code and returns a consolidated list of matching products, prices, and availability from the data provided by the first-party services.

In addition, A. Datum Corporation's service allows consumers to specify both a timeframe and a price range. The composite service monitors the price of the specified product during the timeframe

Figure 1: A simple composite service scenario.

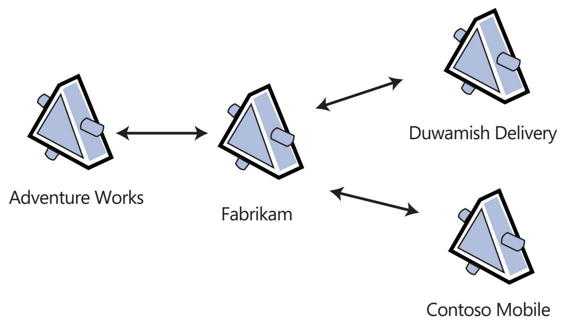
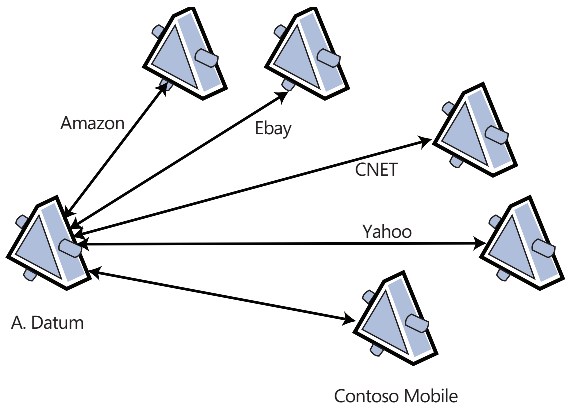


Figure 2: Popular product-related service providers.



identified, and when the price and/or availability changed within that timeframe, a notification would be sent to the consumer.

As the supply of high-demand items can deplete quickly, this service should interact with consumers and provide these notifications across different touchpoints (email, SMS text message, etc.). A. Datum will utilize its own email servers, but for text messaging will leverage Contoso Mobile's text-messaging service.

How to Refer to Customer(s) In a World of Composite Services

If composite services reach their potential, an interesting tree of customers will emerge. Services will be composited into new composite services, which themselves will be composited. When identifying the customers of a service, one must have a flexible means of referencing where a customer sits in the hierarchy and be able to reflect their position as a customer in relation to the various services in the tree. A common approach is to refer to customers with a designation of Cx, where x identifies the distance of the customer from the service in question.

We will use this approach when referring to the participants in our reference scenarios.

In our first reference scenario, Fabrikam is a C1 of Duwamish Delivery and Contoso Mobile. AdventureWorks is a C1 of Fabrikam. AdventureWorks would be seen as a C2 of Duwamish Delivery and Contoso Mobile.

In our second scenario, A.Datum would be the C1 of Amazon, Ebay, Yahoo, CNET, et al. Consumers of the composite service would be seen as a C1 of A.Datum and a C2 of the first-party service providers.

Now that we have a lingua franca for discussing the customers in the composite services n-level hierarchy, let's look at considerations for the services themselves.

More Services Will Become Long Running and Stateful

It is common that services are designed as stateless, and typically invoked with either a request/response or a one-way pattern. As you move to a world of cloud services and distributed systems, you'll see more and more services become long running and stateful, with "services" oftentimes now being an interface to a workflow or business process.

Both of our reference scenarios reflect this. In the first scenario, a package-tracking service monitors the state of package until such point as it's delivered. Depending on the delivery option chosen, this process runs for a minimum of a day, but could run for multiple days, even weeks, as the package makes its way to its destination. In our second reference scenario, the elapsed time could be even longer – weeks, months, even years.

When moving the focus from short-lived, stateless service interactions to these stateful, longer running services, there are a number of important things to keep in mind when designing your systems.

Implement a Publish and Subscribe Mechanism to Effectively Manage Resources and Costs

One of these considerations is the importance of the ability to support subscriptions and publications.

Many long-running, stateful services need to relay information to their service consumers at some point in their lifetime. You'll want to design your services and systems to minimize frequent polling of your service, which unnecessarily places a higher load on your server(s) and increases your bandwidth costs.

By allowing consumers to submit a subscription for notifications, you can publish acknowledgments, status messages, updates, or requests for action as appropriate. In addition to bandwidth considerations, as you start building composites from metered services, the costs associated with not implementing a publish and subscribe mechanisms rise.

In reference scenario 2, the service is querying back-end ecommerce sites for the availability of an item. If a consumer queries this service for a DVD that won't be released until next year, it's a waste of resources for the consumer to query this service (and consequently the service querying the back-end providers) regularly for the next 365 days..

Scenario 2 also highlights that if your service communicates via mechanisms affiliated with consumer devices (i.e., SMS text message or voice applications), the value of publishing to your C1 is minimal – you'll want to allow your C1 customers to subscribe to notifications not just for themselves but also for their customers.

Designing your services to support publications and subscriptions is important in developing long-running services.

Identify How You Will Communicate to Consumers

One of the great things about moving processes to services in the cloud is that the processes can run 24 hours a day, 7 days a week for years at a time. They can be constantly monitoring, interacting, and participating on your behalf.

Eventually, the service will need to interact with a consumer, be it to return an immediate response, an update, or a request to interact with the underlying workflow of a service.

The reality is that while your services may be running 24/7 in the same place, your consumer may not be. Once you've created a publish and subscribe mechanism, you should consider how and when to communicate. Will you interact only when an end consumer's laptop is open? Will you send alerts via email, SMS text message, RSS feed, a voice interface?

Price. Convenience. Control. Be Aware of Your Audience's Sensitivities

When designing your service, you should do so with the understanding that when you transition to the pay-as-you-go cloud, your customers may range from a couple of guys in a garage to the upper echelons of the Fortune 500. The former may be interested in your service, but priced out of using it due to the associated costs for communication services like SMS or voice interfaces. The latter group may not want to pay for your premium services, and instead utilize their own vendors. And there are a large group of organizations in between, which would be happy to pay for the convenience.

When designing your solutions, be sure to consider the broad potential audience for your offering. If you provide basic and premium versions of your service, you'll be able to readily adapt to audiences' sensitivity, be it price, convenience, or control.

White Labeling Communications – Identify Who Will Be Communicating

Per Wikipedia, "A white label product or service is a product or service produced by one company (the producer) that other companies (the marketers) rebrand to make it appear as if they made it."

While perhaps the writer of the entry did not have Web services in mind when writing the article, it is definitely applicable. In a world of composite services, your service consumer may be your C1 but the end consumer is really a C9 at the end of a long chain of service compositions. For those that do choose to have your service perform communications on their behalf, white labeling should be a consideration.

In our first reference scenario, the service providing the communication mechanism is Contoso Mobile. That service is consumed by Fabrikam who offers a composite service to AdventureWorks Web site. Should Contoso Mobile, Fabrikam, AdventureWorks, or a combination brand the messaging?

When communicating to the end consumer, you will need to decide whether to architect your service to include branding or other customization by your C1. This decision initiates at the communication service provider, and if offered, is subsequently provided as the service is consumed and composited downstream.

Is the "Hotmail Model" Appropriate for Your Service?

When looking at whether or not to white label a service, you may want to consider a hybrid model where you allow for some customization by your C1 but still include your own branding in the outbound communications. In our first reference scenario, this could take the form of each text message ending with "Delivered by Contoso Mobile."

I refer to this as the "Hotmail model." When the free email service Hotmail was introduced, it included a message at the bottom of each mail that identified that the message was sent via Hotmail. There is definitely a category of customers willing to allow this messaging for a discounted version of a premium service.

Communicating When Appropriate with Business Rules

Once you've decided how these services will communicate and who will have control over the messaging, you need to determine when the communication will occur and which communication mode is appropriate at a given point of time.

Interaction no longer ends when a desktop client or a Web browser is closed, so you'll want to allow your C1 to specify which modes of communication should be used and when. This can take the form of a simple rule or a set of nested rules that define an escalation path, such as:

```
communicate via email
      if no response for 15 minutes
           communicate via SMS
                if no response for 15 minutes
                     communicate via voice
```

Rules for communication are important. Services are running in data centers and there's no guarantee that those data centers are in the same time zone as the consumer. In our global economy, servers could be on the other side of the world, and you'll want to make sure that you have a set of rules identifying whether it's never appropriate to initiate a phone call at 3am or that it is permitted given a number of circumstances.

If you provide communication mechanisms as part of your service, it will be important to design it such that your C1 can specify business rules that identify the appropriate conditions.

Communicating System Events

As stated previously, long-running, stateful services could run for days, weeks, months, even years. This introduces some interesting considerations, specifically as it relates to communicating and coordinating system events.

When developing your service, consider how you will communicate information down the chain to the consumers of your service and up the chain to the services you consume.

In a scenario where you have a chain of five services, and the third service in the chain needs to shut down (either temporarily or permanently), it should communicate to its consumers that it will be offline for a period of time. You should also provide a mechanism for a consumer to request that you delay the shutdown of your service and notify them of your response.

Similarly, your service should communicate to the services it consumes. Your service should first inspect the state to determine if there is any activity currently ongoing with the service(s) that it consumes, and if so communicate that it will be shut down and to delay the relay of messages for a set period of time.

Design a Mechanism to Easily Retrieve State

There will be times when it is appropriate for an application to poll your service to retrieve status. When designing your service, you'll want to provide an endpoint that serializes state relevant to a customer.

Table 1: Benefits of RSS and ATOM feeds.

Directly consumable by existing clients	A number of clients already have a means for consuming RSS or ATOM feeds. On the Windows platform, common applications including Outlook, Internet Explorer, Vista Sidebar, and Sideshow have support to render RSS and ATOM feeds.
Meta-data benefits	Both RSS and ATOM have elements to identify categories, which can be used to relay information relevant to both software and UI-based consumers. Using this context, these consumers can make assumptions on how to best utilize the state payload contained within the feed.
Easily filterable and aggregatable	State could be delivered as an item in the feed, and those items could be easily aggregated or filtered compositing data from multiple feeds.
Client libraries exist on most platforms	Because RSS and ATOM are commonly used on the Web, there are client-side libraries readily available on most platforms.

Two formats to consider for this are RSS and ATOM feeds. These provide a number of key benefits, as specified in Table 1.

Designing to Fail Over and Not Fall Over

As soon as you take a dependency on a third-party service, you introduce a point of failure that is directly out of your control. Unlike software libraries where there is a fixed binary, third-party services can and do change without your involvement or knowledge.

As no provider promises 100 percent uptime, if you're building a composite service, you will want to identify comparable services for those that you consume and incorporate them into your architecture as a backup.

If, for example, Contoso Mobile's SMS service was to go offline, it would be helpful to be able to reroute requests to a third-party service, say A.Datum Communications. While the interfaces to Contoso Mobile's and A. Data Communication's services may be different, if they both use a common data contract to describe text messages, it should be fairly straightforward to include failover in the architecture.

Consider Generic Contracts

In the second reference scenario, for example, eBay and Amazon are both data sources. While each of these APIs has result sets that include a list of products, the data structures that represent these products are different. In some cases, elements with the same or similar names have both different descriptions and data types.

This poses an issue not just for the aggregation of result sets, but also for the ability of a service consumer to seamlessly switch over from Vendor A's service to that of another vendor's service. For the service consumer, it's not just about vendor lock-in, but the ability to confidently build a solution that can readily failover to another provider. Whether a startup like Twitter or a Web 1.0 heavyweight like Amazon, vendor's services do go offline. And an offline service impacts the business' bottom line.

Generic contracts — for both data and service definitions — provide benefits from a consumption and aggregation standpoint, and would provide value in both of our reference scenarios. Service providers may be hesitant to utilize generic contracts, as they make it easier for customers to leave their service. The reality is that while generic contracts - for both data and service definitions - do make it easier for a customer to switch, serious applications will want to have a secondary service as a backup. For those new to market, this should be seen as an opportunity and generic contracts should be incorporated into your design.

Handling Service Failure

Many service providers today maintain a Web page that supplies service status and/or a separate service that allows service consumers to identify whether a service is online. This is a nice (read: no debugger required) sanity check that your consumers can use to determine whether your service is really offline or if the issue lies in their code.

If you're offering composite services, you may want to provide a lower level of granularity, which provides transparency not just on your service, but also on the underlying services you consume.

There is also an opportunity to enhance the standard status service design to allow individuals to subscribe to alerts on the status of a service. This allows a status service to proactively notify a consumer when a service may go offline and again when it returns to service. This is particularly important to allow your consumers to proactively handle failovers, enabling some robust system-to-system interactions.

This can provide some significant nontechnical benefits as well, the biggest of which is a reduced number of support calls. This can also help mitigate damage to your brand – if your service depends on a third-party service, say Contoso Mobile, your customers may be more understanding if they understand the issue is outside of your control.

Which Customer(s) Do You Need to Know About?

For commercial services, you will, of course, want to identify your C1 consumers and provide a mechanism to authenticate them. Consider whether you need to capture the identities of customers beyond C1 to C2 to C3 to Cx.

Why would you want to do this? Two common scenarios: compliance and payment.

If your service is subject to government compliance laws, you may need to audit who is making use of your service. Do you need to identify the consumer of your service, the end-consumer, or the hierarchy of consumers in between?

In regards to payment, consider our first reference scenario. Duwamish and Contoso sell direct to Fabrikam, and Fabrikam sells direct to Adventureworks. It seems straightforward, but what if Fabrikam is paid by AdventureWorks but decides not to pay Duwamish and Contoso? This is a simple composition, unless you expand the tree, with another company that creates a shopping cart service that consumes the Fabrikam service, and is itself consumed in several e-commerce applications.

While you can easily shut off the service to a C1 customer, you may want to design your service to selectively shut down incoming traffic. The reality is that in a world of hosted services, the barriers to entry for switching services should be lower. Shutting off a C1 customer may result in C2 customers migrating to a new service provider out of necessity.

For example, if Fabrikam also sells its service to Amazon, Contoso Mobile and Duwamish may recognize that shutting off service to C2 customer Amazon could drive it to select a new C1, significantly cutting into projected future revenues for both Contoso Mobile and Duwamish. AdventureWorks, however, is a different story and doesn't drive significant traffic. Both Contoso and Duwamish could decide to reject service calls from AdventureWorks but allow service calls from Amazon while they work through their financial dispute with Fabrikam.

Conclusion

There is no doubt that now is an exciting time to be designing software, particularly in the space of distributed systems. This opens up a number of opportunities and also requires a series of considerations. This article should be viewed as an introduction to some of these considerations, but is not an exhaustive list. The goal here was to introduce some of these considerations, enough to serve as a catalyst to drive further conversation. Be sure to visit the forums on the *Architecture Journal* site to continue the conversation.

About the Author

Marc Mercuri is a principal architect on the Platform Architecture Team, whose recent projects include the public-facing Web sites Tafiti.com and RoboChamps.com. Marc has been in the software industry for 15 years and is the author of *Beginning Information Cards and CardSpace: From Novice to Professional*, and co-authored the books *Windows Communication Foundation Unleashed!* and Windows Communication Foundation: *Hands On!*

Architectural Patterns for Distributed Computing

by Joshy Joseph, Jason Hogg, Dmitri Ossipov, Massimo Mascaro, and Danny Garber

Summary

As the technologies supporting design, development, and deployment of distributed applications continue to advance, it is more important than ever for architects and developers to establish stable libraries of architectural patterns based on recurrent experiences developing such distributed solutions. This article will explore patterns harvested from customer engagements—mapping real-world scenarios to patterns and then describing how said patterns were implemented on current technologies. Our analysis also includes a discussion of the forces leading to the application of a particular pattern and the resulting context that must be considered.

Introduction

Distributed systems architecture has rapidly evolved by applying and combining different architecture patterns and styles. In recent years, decoupling interfaces and implementation, scalable hosting models, and service orientation were the prominent tenets of building distributed systems. Distributed systems have grown from independent enterprise applications to Internet connected networks of managed services, hosted on-premise and/or clouds.

Creating successful distributed systems requires that you address how that system is designed, developed, maintained, supported, and governed. Microsoft's Service Oriented Architecture Maturity Model (SOAMM) assessment method helps customers assess maturity elements and prioritize the evolution of their enterprise architecture. Organizations are applying methods, patterns, and technologies to model systems and enable proper governance.

We will discuss patterns harvested from different consulting engagements and examine how these patterns were implemented on current technology offerings from Microsoft. Building distributed systems involves three core capabilities based on these roles:

1. implementation of applications/services;
2. consumption of services from within and external to organizations; and
3. administration of services or composite services.

Implementation of Services and Business Processes

Let's start with the implementation capabilities needed to build and expose services. Implementation capabilities describe an organization's methods to implement effective best practices, identify and apply patterns, and provide services.

Service Design

Five years ago, the Patterns & Practices (P&P) team was asked to create a reference application for developing distributed applications to accompany the release of the second version of .NET. To design this reference application, P&P spent much time talking with customers and consultants in the field to understand the challenges they encountered when designing, developing, and deploying applications. As a result of these interactions, P&P identified recurring patterns relevant to SOA from narrative-based pattern guides culminating with the Web Service Software Factory.

Patterns. An early promise of services was support for interoperability within a heterogeneous environment, but given the wide range of technologies and emerging nature of many of the WS* standards, it was critical for the service's business logic to be completely independent of the service's interface, allowing the bindings and policies associated with the service to evolve separately. Furthermore, the actual technology used to implement the services could also minimize the impact on business logic. The Service Interface pattern described a means of designing services to cater to these forces.

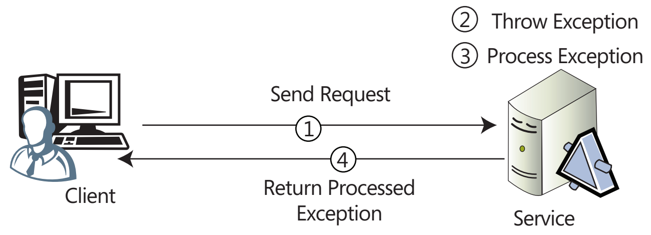
In addition to separating the design of the service interface from the

"The goal of any IT system is to implement the logic behind an organization's business processes. As organizations automate more and more processes, the IT department's perceived value grows, as well as its responsibilities on the business processes realization itself. This is particularly true in industries where the products are materialized as IT systems, such as areas of commerce and trading, banking, and insurance."

service's implementation, cross-cutting concerns such as authentication, authorization, message validation, and auditing were implemented using a pipeline similar to the Pipes and Filters pattern. This pipeline allowed these cross-cutting concerns to be specified declaratively independent of the service's business logic. This pattern is used within the WSSF to ensure that cross-cutting concerns such as validation are configured declaratively.

A core tenant of SOA is the concept that a service's boundaries are explicit — interactions are performed using well-defined messages. Furthermore, the service must ensure that such messages provided

Figure 1: Exception Shielding pattern



as input are validated because incoming data may be malformed and transmitted for malicious reasons. The service must also ensure that erroneous behaviors occurring within the service related to the internal implementation of the service are not accidentally leaked. The Message Validator pattern describes several approaches that can validate incoming messages prior to execution of business logic. The WSSF actually uses the Enterprise Library Validation Block to support these requirements. A second pattern called the Exception Shielding pattern describes precautions that a service author can take to ensure that exceptions are caught and sanitized prior to returning a fault to a client — while still ensuring that sufficient information is logged for administrators to troubleshoot. Figure 1 illustrates this patterns interaction.

Finally, when it came time to deploy these early services to a production environment one challenge that many customers faced was that the operations folks would not allow the clients to talk directly to the Web service. For customers with external clients, it was against company policy to have an external application interact directly with an application residing on an internal network. The solution was to deploy a Perimeter Service Router into the perimeter network, allowing it to take responsibility for directing incoming messages to appropriate internal resources. Figure 2 illustrates this scenario.

Process Design

The goal of any IT system is to implement the logic behind an organization's business processes. As organizations automate more and more processes, the IT department's perceived value grows, as well as its responsibilities on the business processes realization itself.

This is particularly true in industries where the products are materialized as IT systems, such as areas of commerce and trading, banking, and insurance. An IT department's effective implementation of the business process can lead to predominant market positions and impressive returns. In turn, IT inertias and the inability to execute quickly can lead to business disasters.

Cost and time important considerations when designing and implementing business processes is the need to optimize cost and time. To reduce costs, reuse effective patterns, software factories, and agile/test-driven software methodologies. Noncompliance with specifications and changing business processes and policies during the course of the implementation contribute to delays.

Business-critical processes must ensure process reliability and availability and enable governance.

When dealing with reuse effectiveness, the graphical composition of complex processes from small reusable building blocks and the augmentation of those technologies with domain-specific extensions showed greater value. A good system will allow for process templating; for example, the ability to create a macro process template that derives a number of processes that follow the same template. The usage of graphical composition tools allows non developer users to review and change processes. In order to run a process, you need a hosting environment, and the ability to locate and read the process definition and efficiently execute it. In addition, environments should provide reliability, fault tolerance, on demand migration to different environments, and transactionality.

Scenario Description. We are involved with a banking automation project in Europe. This bank is active in the retail market with a large diffusion of branches across the territory, and is gaining a lot of traction from its customer-directed financial products.
The Bank management has opted into a strategy based on four pillars:

- Maintain customer contact but steer the branch from a mere teller office to a financial services shop.
- Boost self-service teller operations across all the bank delivery channels (including ATMs).
- Invest in cross-selling and customer-centric offerings at any customer's point of contact based on context.
- Enlarge the customer channel reach boosting call center, mobile, and intermediary sales of financial products.

This bank is currently in the process of rebuilding its main branch application through a new business process using SOA methods.

Key Functional and Nonfunctional Characteristics. In our experience, effective business process design and execution infrastructure should provision:

- Composition of business processes and integration of backend systems
- Smart client interaction with the services' infrastructure, composition, and frontend/backend integration
- Conversational capabilities of processes
- Shortcut the analyst -developer interaction with process-driven development
- Reduce the impact of change requests
- Online/offline support at the process level; the ability to relocate business processes out of the center

Figure 2: Perimeter Service Router pattern

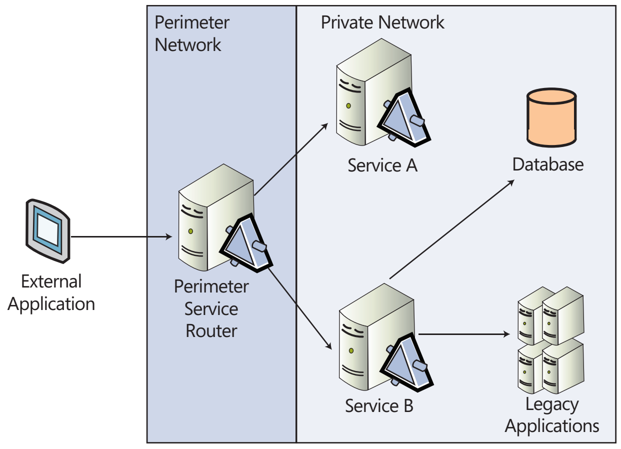


Table 1 : Business Process Workflow Design Patterns category

Patterns Category	Problems	Solution
Process driven development	How to develop business process conforming to the well-known workflow patterns. How to quickly adapt to process change requirements.	• Using extensible graphical workflow systems to build domain-specific environments, such as WF • Workflow patterns provide a set of patterns for control-flow, resource, data, and exception handling • DCS Software factory toolkit automates known patterns such as process templating, process interception, context-based process selection, and stateful interactions • Externalizing business rules from workflow using WF Rule engine or BizTalk Rule Engine
Separation of contracts and implementations	How to ensure the same service contracts can be implemented in multiple fashions.	• Use WCF contract extendibility features • Use MSE to virtualize different services and re-expose them with a unified contract • DCS Task Factory application toolkit provides a framework to dynamically choose the implementation
Transparent process migration and resilience	How to move processes seamless among the machines that need to host them, to handle disconnected scenarios and faults.	• Use DCS WF hosting capabilities to persist/move workflows
Composite applications and services	How to write client applications (across multiple channels) to quickly compose existing UI components and services in different ways.	• Build and host services using ESB Guidance Toolkit • Use DCS Software Factory toolkit • Use either the CAB or Prism to build composite smart clients • Use Customer Care Framework (CCF) 2009 to build frontends and backends (it embeds DCS) with composite capabilities spanning multiple channels and WF activated services
Conversational processes	How to allow a process to maintain its state across multiple calls coming from one or more caller.	• Use BizTalk orchestration exposed as services • Use WF activated services in .NET 3.5 and WF standard persistence • Use DCS to build persistent, WF designed logic
Stateful interceptors	How to intercept service calls transparently, modifying their behavior via stateful interceptors able to correlate multiple operations; for example, to apply new marketing-driven, customer state-specific behaviors.	• Use DCS business Transparent Processes and DCS hosting engine
Process status monitoring, business activity monitoring	How to monitor the current state of a process or business activity.	• Use BizTalk BAM to monitor either the BizTalk orchestrations or the WF hosted services (via BTS BAM interceptor) • Use default tracking capabilities in workflow foundation (no OOB provision for intraprocess monitoring)

- Large scale deployment, configuration, and management
- Separate rules from process code for maximum flexibility.

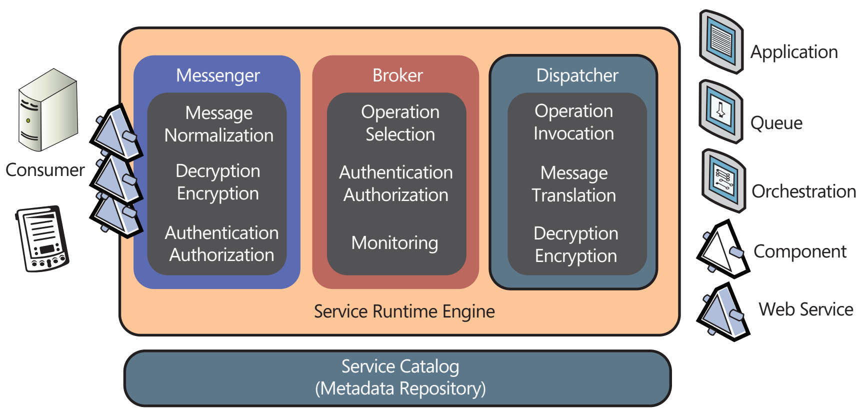
Patterns. We gathered patterns related to business process development that illustrate a collection of architectural and workflow patterns:

- Basic Workflow Patterns: control-flow, resource, data, and exception-handling related patterns.
- Advanced Workflow Patterns: wizard-based tools that drive contract-first development, inclusion of transparent processes or intermediaries, context-based process selection, enabling stateful interactions, and externalization of business rules. Some of these patterns are not formalized elsewhere.
- Workflow Hosting Patterns: a flexible hosting platform exposing workflows as web services, locating and selecting business processes from a repository based on context, scalable process execution environment, and pluggable workflow runtimes.
- Business Activity Monitoring: monitoring the current state of a business activity.

These patterns force you to build service compositions with the flexibility to dynamically change the activities through an appropriate service implementation based on business rules.

Technologies Realizing Patterns. Microsoft offers two technology solutions to address business process development and hosting: BizTalk server and Windows Workflow (WF). BizTalk offers a platform and tools to connect with services inside and outside of organizations, including a large number of adapters to legacy systems. WF is the programming model, engine, and tools for quickly building workflow-enabled applications. In addition, Microsoft consulting developed a wizard-driven toolkit and runtime, Distributed Connectivity Service (DCS), to provide guidance on developing and hosting business processes based on the aforementioned patterns. DCS provides service management capabilities such as service location transparency, dynamic clients, context-driven business process selection, scalable hosting of workflows, and a security token service for exposing business processes as services. Depending on the enterprise scenario and hosting requirements, you could select BizTalk, DCS, or a combination. Table 1 lists the patterns category, along with the problem statements and associated technology implementations.

Figure 3: The Managed Services Engine



Deploying Service

Deploying services to a production environment is complicated. Operational staffs need to decide how the service should be exposed, who has access to the service, which protocols should be supported, and how to version a service. In some cases it is against company policy to have an external application interact directly with an application residing on an internal network. Some customers restrict service access to a specific set of credentials. Organizations would like to expose multiple versions of the service for old and newer clients. Services may use legacy protocols that clients don't support.

Patterns. While addressing these scenarios, numerous patterns emerged:

- Deploying a Perimeter Service Router into the network allows it to take responsibility for directing incoming messages to appropriate internal resources.
- Service Catalog Pattern: Extracts and stores service-related metadata (policy, endpoint information, protocol binding/channels, service/client behaviors).
- Service Virtualization: Consolidates a single service or multiple services to a virtualized service.
- Service Versioning: Enables multiple versions of the same service to deploy simultaneously.
- Service Activity Monitoring: A central hub monitors client interaction with the service.

These patterns resulted in a technology framework called Managed Services Engine. As shown in Figure 3, MSE provides a message hub with message normalization, brokering, and dispatching capabilities.

Consumption of Services

The consumption of services from within the enterprise (as well as those external to it) requires business enablement and platform capabilities. These capabilities help enterprises to effectively adopt and promote the use of services by providing a foundation to support and enhance the consumption of enterprise services by others.

Enterprise Service Bus. The term Enterprise Service Bus (ESB) is widely used in the context of implementing an infrastructure for enabling a service-oriented architecture (SOA). An ESB product supports a variety of patterns, hence by breaking various ESB's down into their constituent patterns, you get a real sense of the capabilities implemented within each solution.

Scenario Description. We worked with a major insurance provider that used the BizTalk server as a platform choice for the integration of legacy systems. Recently, this company became a provider of auto claim services. Receiving an auto claim from claim office, creating the claim process, and sending the claim to an independent adjuster for damage inspection and receiving a repair estimate for approval, parts procurement, and further repairs is a common scenario. These imply configurable policy-based service mediation and composition of enterprise and third-party services, thereby coordinating the execution of distributed business processes.

Key Functional and Nonfunctional Characteristics. Key functional requirements to define the architecture included:

- Loosely coupled messaging environment and protocol mediation
- Registry-driven resolution of business service endpoints
- Itinerary-based message routing
- Service life cycle management and dynamic business changes
- Monitoring the health of individual and composed services
- Enabling automated service provisioning to service registry.

Patterns. You can think of an ESB as a collection of architectural patterns based on traditional enterprise application integration (EAI) patterns, message-oriented middleware, web services, interoperability, host system integration, and interoperability with service registries and repositories.

Applying message routing patterns forces you to build service compositions with flexibility in a dynamically evolving enterprise:

- Routing Slip Pattern —routing a message through consecutive number of steps when the sequence of steps is not known and may vary for each message
- Recipient List Pattern — routing a message to a dynamic list of recipients
- Content-Based Router—routing a message to its destination based on message content
- Scatter-Gather — maintaining the overall message flow when a message must be sent to multiple recipients, where each may send a reply
- Repair and Resubmit — modifying content of externalized fault that contains the original message and resubmitting to resume the processing sequence

Applying message transformation patterns forces you to use canonical data models to minimize intra-application dependencies when different data formats are being used:

- Data Transformation — transform a message from one format to another when the format is semantically equivalent

"Increasingly, the Internet is being used as the delivery vehicle for innovative new service offerings such as Web 2.0. Businesses are developing bundled service offerings to provide value-added services to their customers over the Internet. These service offerings increasingly rely on the ability to aggregate multiple composite services from a wide variety of different sources."

- Content Enricher —communicate with another system to include additional information in the message content.

Technologies Realizing Patterns. To implement best practices for building service compositions, developers and architects should clearly articulate the following:

- Identify a baseline architecture that provides a view of an executable slice through the overall system that is designed to implement the significant use cases
- Choose integration topology that suits most of the known requirements
- Incorporate functional scenarios into baseline architecture by mapping the design into a set of known integration patterns.

Enterprise Service Bus Guidance (ESBG) for Microsoft BizTalk Server R2 provides best practices for implementing ESB usage patterns using the Microsoft platform.

Table 2 lists the patterns category along with the respective problems and solutions.

Patterns Addressing Internet Service Bus (ISB) Scenarios. Internet Service Bus (ISB) addresses cross enterprise service collaboration scenarios. Organizations attempting to implement such scenarios with a traditional web services composition/ESB approach are facing challenges when it comes to spanning business processes across the enterprise boundaries.

In this section, we offer a design pattern, Agent Traveling Design Pattern, and specifically two of its subsidiary patterns, Itinerary Pattern and Ticket Forward Pattern, which can help to solve these challenges. Traveling patterns deal with various aspects of managing the movements of mobile agents, such as routing and quality of service. Examples of the Itinerary pattern can be found in various implementations of ESB.

We will demonstrate how with help of MSE and ESBG one can achieve the desired Itinerary pattern architecture that spans network and geographic boundaries.

Scenario Description. We are involved with a medical insurance provider that decided to embark on SOA to modernize its Information technology while leveraging existing legacy data centers and web services deployed across the nation. Its main data center is connected in a cross-star network to several satellite data centers as well as to the third-party partner's medical claims management agencies.

This provider decided to embrace all of its data center services and LOB applications into a set of workflow processes, utilizing the Itinerary pattern, which is dynamically controlled by the business rules, runtime policies, and SLAs.

Key Functional and Nonfunctional Characteristics. The requirements included:

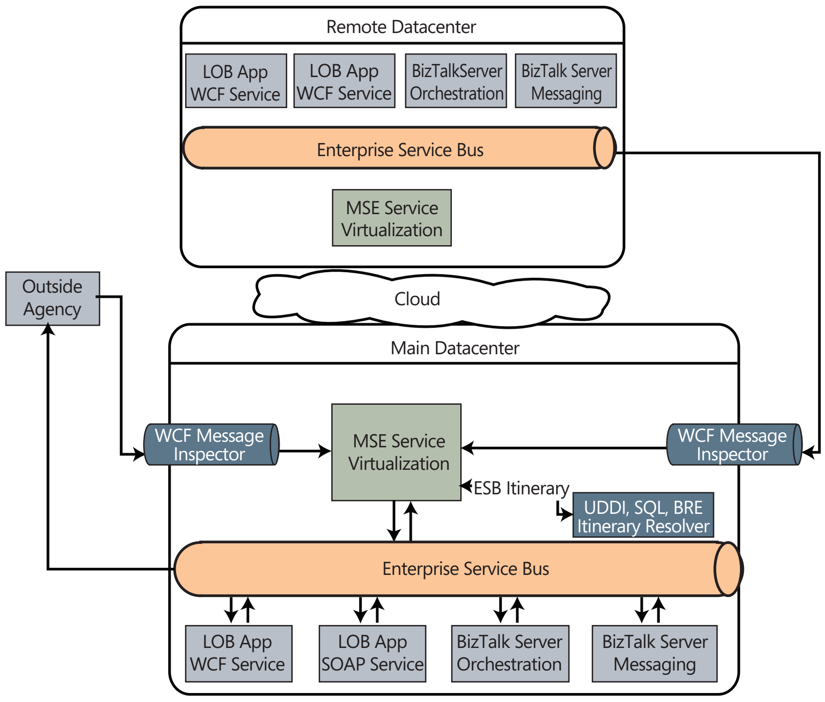
- Leverage legacy data center systems while creating new services
- Support multiple communications protocols (WCF, SOAP, REST, etc.)
- Operate in an environment with several data centers, geographically dispersed
- Implement multiple itinerary workflows, some of them nested in each other
- Provide the ability for real-time monitoring and routing of transactions.

Patterns. For evaluation purposes, we selected the Itinerary-Ticket pattern for this customer scenario, which is a combination of two traveling patterns: Itinerary pattern and Ticket pattern. The Itinerary pattern is an example of a traveling pattern that is concerned with routing among multiple destinations. An itinerary maintains a list of

Table 2: Service Composition Patterns categories

Patterns Category	Problems	Solution
Message Routing Patterns	How to define service compositions. How to enforce development best practices for service discovery.	• Use the Routing Slip pattern for implementing itinerary-based routing. • Leverage key itinerary abstractions to compose messaging and orchestrations into a sequence of itinerary steps.
	How to force the reuse of complex messaging scenarios independently from service contracts.	• Use service registry and categorization schemas for service discovery. • Define service orchestrations as reusable components. These components can be incorporated as reusable itinerary steps.
Message Transformation Patterns	How to apply data model transformations dynamically.	• Define data model transformations to be executed as part of itinerary-based routing.
Operations Management	How to troubleshoot messaging activities, repair, and resubmit a message. How to track execution of itinerary-based routing.	• Apply the Repair and Resubmit pattern by providing infrastructures with capabilities of externalizing faults to a centralized store and use management applications for generating alerts, message editing, and resubmission to a service endpoint when applicable. • Use BizTalk BAM to track message interchanges and execution history of itinerary steps.

Figure 4: The interaction of BizTalk Server, MSE, and ESB Guidance components



destinations, defines a routing scheme, handles special cases such as what to do if a destination does not exist, and always knows where to go next. Objectifying the itinerary allows you to save it and reuse it later. The Ticket pattern is an enriched version of a URL that embodies requirements concerning quality of service, permissions, and other data. For example, it may include time-out information for dispatching an agent to a remote data center in our customer scenario. It can also contain the security and auditing metadata required by the regulations and controlled (injected) by the execution policies. Thus, instead of naively trying to dispatch to a disconnected host forever, the agent now has the necessary information to make reasonable decisions while travelling. An agent is capable of navigating itself independently to multiple hosts. Specifically, it should be able to handle exceptions such as unknown hosts while trying to dispatch itself to new destinations or make a composite tour (e.g., return to destinations it has already visited). It might even need to modify its itinerary dynamically. Consequently, it is probably preferable to separate the handling of navigation from the agent's behavior and message handling, thus promoting modularity of every part.

We developed an architecture consisting of BizTalk Server, MSE, and ESB Guidance components. In this case, all three of Microsoft's SOA technologies were brought into play to provide the required functionality. Figure 4 illustrates the interaction of these components.

Service Administration

Administration covers systems management, governance, and operational aspects of distributed systems. Most distributed system implementations today are referred to as "well understood and well behaved." These implementations assume that the services being used are in a controlled environment, usually in a single data center or across a trusted partner channel. Increasingly, the Internet is being used as the delivery vehicle for innovative new service offerings such as Web 2.0. Businesses are developing bundled service offerings to provide value-added services to their customers over the Internet. These service offerings increasingly rely on the ability to aggregate multiple composite services from a wide variety of different sources. The challenges include service-level management, resource management, service provisioning, and monitoring. This mandates services to follow the principles of service enablement and supports methods for discovery, provisioning, monitoring, security, and usage measurements. These management features are similar to the well-known telecommunication patterns of Fault, Configuration, Accounting, Performance, and Security (FCAPS).

Scenario Description. A few years back we were involved with a telecommunication service provider to automate its service delivery platform infrastructure. This company leveraged web services capabilities to develop and expose its core service offerings and reused services from other service providers such as geo-location providers, ring-tone providers, and credit verification systems; for example, bundle a ring-tone download service from a public service provider along with network services and generate billing information for each successful download. This system is always available and the provider monitors the usage of the third-party services to validate the SLA.

Key Functional and Nonfunctional Characteristics. Key functional requirements included:

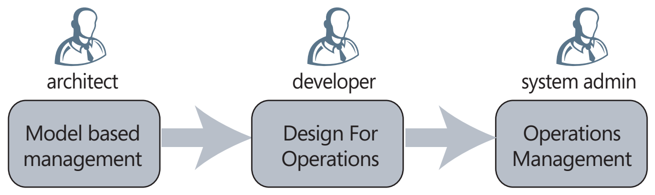
- Developing and deploying services for manageability and operations-friendly
- Making services easily discoverable and composable
- Monitoring the health of individual and composed services
- Collecting metrics on service usage and performance
- Enabling automated provisioning of services.

Patterns. This experience helped us discover a series of patterns related to service management. These patterns and best practices have a deep-rooted background in building telecommunication network best practices, which is considered the most complex distributed computing network with highly distributed and highly reliable systems.

Operations-Friendly Service Modeling Patterns. It is a best practice to adopt the principles around knowledge-driven management while modeling services or composite services:

- Exceptions are unavoidable: Always accept failures and build systems to adapt and provide recovery methods. Also, show exceptions as needed rather than flooding systems with the same or related exceptions.
- Collecting metrics and logs: Collecting information regarding failures, performance measures, and usage will help identity the failure conditions and service-level management.

Figure 5: Role-based management instrumentation



- Automate everything: People and processes are error prone. Automation will help avoid human errors. For example, "Minimize Human Intervention" pattern discusses this in the context of telecommunication networks.
- Complex environment factors: Distributed systems needs to collaborate with services from different domains, handle heterogeneous devices, complex configurations, and a large number of users. Consider factors around latency, network failures, and service-level agreements.

Analysts and architects should consider business and operational behaviors and model individual services, business processes, and service aggregates. These behaviors are documented as part of service models and communicated to developers to facilitate instrumentation. Standards are emerging to codify these requirements and constraints including Common Information Models (CIM), Service Modeling Language (SML), and Web Service Distributed Management (WSDM).

Design for Operations Patterns. Instrumenting services as operations-friendly is complex. Developers should instrument services and business processes with health events, measurement points, performance metrics, provisioning interfaces, and operational support metrics. These patterns are emerging to provide better guidance on how to instrument systems for manageability.

Operational Management Patterns. Operational systems need management models, key performance indicators (KPI), and associated events to measure operational intelligence. These systems resemble the observer-event patterns or control bus pattern to define the interactions of managed entities (e.g., web service) and management servers (e.g., System Center Operations Manager). For example, Management Packs will help codify the system requirements, correlation methods, and automation steps. SCOM will push or pull data from web services to understand service health. Operational systems should monitor continuous availability, performance, and degrading conditions of service. Systems should provide capabilities to aggregate log files, send health-check events, and service failover methods. Standards such as WS-Management provide a way for systems to access and exchange management information.

Technologies Realizing Patterns. A service life cycle management practice:

- Creates a management model for distributed applications
- Designs and implements services for operations-friendliness
- Develops and deploys management packs, performance counters, and KPI models
- Monitors the performance, availability, and response times of services
- Audits service messages and enables secure transactions
- Enables business activity monitoring to manage service-level constraints
- Automates management systems to respond errors, violations, and generate notifications.

As shown in Figure 5, an architect defines management models based on the system's operational requirements, developers add code instrumentation, and systems administers look for events and patterns to take necessary actions.

Table 3 lists the patterns categories in the service management domain.

Conclusion

In this article, we discussed a number of patterns that have been observed during customer engagements focused on the development of distributed systems. Given our customers increased dependence on technology and the ever-increasing rate at which technology is

Table 3: Service Management Patterns categories

Patterns Category	Problems	Solution
Operations-Friendly Service Modeling Pattern	How to collect nonfunctional and operational requirements/constraints and document these requirements.	• Use modeling tools such as Visual Studio Architect Modeler • Use System Center Management Model Designer Power Tools
Design for Operations Patterns	How to instrument in code for service manageability.	• Develop services using well-enabled service principles using WES Toolkit • Develop instrumentation using Design For Operations tools
Operational Management Patterns	How to enable services or composites to be well-behaved. How to monitor business activities.	• Use SCOM for management • Use SCOM Console to author/configure management packs • MSE to collect system messages and events • Enable BizTalk BAM for activity monitoring

evolving, it is in our opinion that it is more and more critical to ground how we design and describe our products in patterns. Without doing this, we increase the perceived complexity of moving from a legacy technology to a newer technology. The patterns described here were implemented in offerings such as DCS, ESBG, DCS, and the WSSF — all of which are built on top of WCF. When Oslo is released, these same patterns will continue to be valid with substantially lower cost.

We must take responsibility for capturing and sharing recurrent solutions to problems such that our solutions and products can provide better out-of-the-box support for core distributed computing patterns. If you are interested in learning more about how to do this, please contact the authors, who are in the process of developing such a library of patterns.

The authors would like to thank Blair Shaw and William Oellermann for reviewing an early draft of this article.

Resources

"Agent Design Patterns: Elements of Agent Application Design," by Yariv Aridor and Danny B. Lange
http://www.moe-lange.com/danny/docs/patterns.pdf

Assessment and Roadmap for Service Oriented Architecture
http://download.microsoft.com/download/9/d/1/9d1b5243-21f6-4155-8a95-1f52e3c
aeeaa/SOA_Assessment-and-Roadmap_Datasheet_2007.pdf

Codeplex
Composite WPF: http://www.codeplex.com/CompositeWPF
Design for Operations: http://www.codeplex.com/dfo
Enterprise Service Bus Guidance Community: http://www.codeplex.com/esb

Managed Services Engine Community: http://www.codeplex.com/servicesengine
Smart Client Guidance: http://www.codeplex.com/smartclient
Web Service Software Factory: http://www.codeplex.com/servicefactory

Design Patterns: Elements of Reusable Object-Oriented Software, by Erich
Gamma et. al
http://www.amazon.com/Design-Patterns-Object-Oriented-Addison-

Wesley-Profess
ional/dp/0201633612

Distributed Management Task Force
Common Information Model: http://www.dmtf.org/standards/cim/
Web Services for Management: http://www.dmtf.org/standards/wsman/

Enterprise Integration Patterns
Control Bus: http://www.integrationpatterns.com/ControlBus.html
Recipient List: http://www.integrationpatterns.com/RecipientList.html
Routing Slip: http://www.integrationpatterns.com/RoutingTable.html

Microsoft BizTalk Server
http://www.microsoft.com/biztalk/en/us/default.aspx
Microsoft Services Providers – Customer Care Framework (CCF)
http://www.microsoft.com/serviceproviders/solutions/ccf.mspx

Microsoft SOA Products: Oslo
http://www.microsoft.com/soa/products/oslo.aspx

Microsoft System Center
http://www.microsoft.com/systemcenter/en/us/default.aspx

MSDN: Biztalk Server 2006: BAM FAQ
http://msdn.microsoft.com/en-us/library/aa972199.aspx

MSDN: Building Services by Using the WES Toolkit
http://msdn.microsoft.com/en-us/library/aa302608.aspx

MSDN: Microsoft patterns & practices
http://msdn.microsoft.com/en-us/library/ms998572.aspx

OASIS Web Services Distributed Managements (WSDM) TC
http://www.oasis-open.org/committees/tc_home.php?wg_
abbrev=wsdm=

Service Modeling Language (SML) Working Group
http://www.w3.org/XML/SML/

Workflow Patterns Initiative
http://www.workflowpatterns.com

Visual Studio Team System 2008 Architecture Edition
http://msdn.microsoft.com/en-us/vsts2008/arch/default.aspx

Visual Studio Team System Management Model Designer Power Tool
http://www.microsoft.com/downloads/details.aspx?FamilyID=7F45F9A9-56F5-4E1F-
9FED-A3E4342FB607&displaylang=en

About the Authors

Joshy Joseph is a principal architect with Microsoft Services Managed Solutions Group. He can be reached at joshy.joseph@microsoft.com.

Jason Hogg is an architect inside the Microsoft Services Managed Solutions Group. He can be reached at jahogg@microsoft.com.

Dmitri Ossipov is a program manager at Microsoft with the patterns & practices team, worked on ESB Guidance for Microsoft BizTalk Server 2006 R2 and the Web Service Software Factory.

Massimo Mascaro is a program manager in the Customer Care Framework group at Microsoft, working on large scale enterprise architecture.

Head in the Cloud, Feet on the Ground

by Eugenio Pace and Gianpaolo Carraro

Summary

Building on the Software as a Service (Saas) momentum, distributed computing is evolving towards a model where cloud-based infrastructure becomes a key player, to the level of having a new breed of companies deploying the entirety of their assets to the cloud. Is this reasonable? Only time will tell. In this article, we will take an enthusiastic yet pragmatic look at the cloud opportunities. In particular, we will propose a model for better deciding what should be pushed to the cloud and what should be kept in-house, explore a few examples of cloud-based infrastructure, and discuss the architectural trade-offs resulting from that usage.

Prelude

From a pure economical perspective, driving a car makes very little sense. It is one of the most expensive ways of moving a unit of mass over a unit of distance. If this is so expensive, compared to virtually all other transportation means (public transport, trains, or even planes), why are so many people driving cars? The answer is quite simple: control. Setting aside the status symbol element of a car, by choosing the car as a means of transportation, a person has full control on when to depart, which route to travel (scenic or highway), and maybe most appealing, the journey can start from the garage to the final destination without having to rely on external parties. Let's contrast that with taking the train. Trains have strict schedules, finite routes, depart and arrive only at train stations, might have loud fellow passengers, are prone to go on strike. But, of course, the train is cheaper and you don't have to drive. So which is one is better? It depends if you are optimizing for cost or for control.

Let's continue this transportation analogy and look at freight trains. Under the right circumstances, typically long distances and bulk freight, transport by rail is more economic and energy efficient than pretty much anything else. For example, on June 21, 2001 a train more than seven kilometers long, comprising 682 ore cars made the Newman-Port Hedland trip in Western Australia. It is hard to beat such economy of scale. It is important to note, however, that this amazing feat was possible only at the expense of two particular elements: choice of destination and type of goods transported. Newman and Port Hedland were the only two cities that this train was capable of transporting ore to and from. Trying to transport ore to any other cities or transporting anything other than bulk would have required a different transportation method.

By restricting the cities that the train can serve and restricting the type of content it can transport, this train was able to carry more than 650 wagons. If the same train was asked to transport both bulk and passengers as well as being able to serve all the cities of the western coast of Australia, the wagon count would have likely be reduced by at least an order of magnitude.

The first key point we learn from the railroads is that high optimization can be achieved through specialization. Another way to think about it is that economy of scale is inversely proportional to the degree of freedom a system has. Restricting the degrees of freedom (specializing the system) achieves economy of scale (optimization).

- Lesson 1: The cloud can be seen as a specialized system with fewer degrees of freedom than the on-premises alternative, but can offer very high economy of scale.

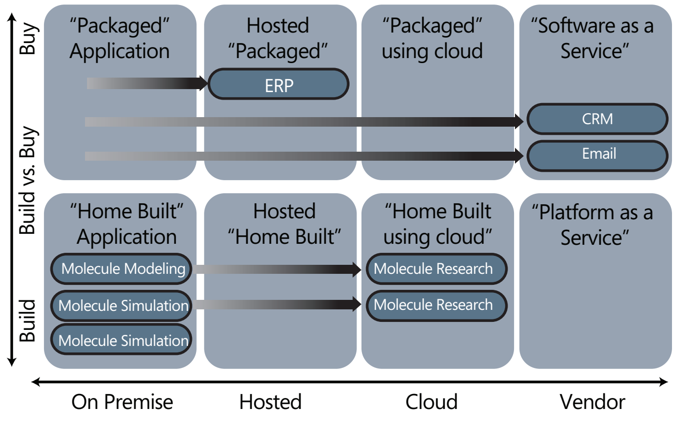
Of course, moving goods from only two places is not very interesting, even if you can do it very efficiently. This is why less optimized but more agile means of transport such as trucks are often used to transport smaller quantities of goods to more destinations.

As demonstrated by post offices around the globe, their delivery network is a hybrid model including high economy of scale — point-to-point transportations (e.g. train between two major cities); medium economy of scale trucks — dispatching mail from the train station to the multiple regional centers; and very low economy of scale but super high flexibility, — delivery people on car, bike, or foot capable of reaching the most remote dwelling.

- Lesson 2: By adopting a hybrid strategy, it is possible to tap into economy of scale where possible while maintaining flexibility and agility where necessary.

The final point we can learn from the railroads is the notion of efficient transloading. Transloading happens when goods are transferred from one means of transport to another; for example, from train to a truck as in the postal service scenario. Since a lot of cost can be involved in transloading when not done efficiently, it is commonly done in transloading hubs, where specialized equipment can facilitate the process. Another important innovation that lowered the transloading costs was the standard shipping container. By packaging all goods in a uniform shipping container, the friction between the different modes of transport was virtually removed and finer grained hybrid transportation networks could be built.

Figure 1: Big Pharma On-Premises vs. Big Pharma leveraging hosted "commodity" services for noncore business capabilities and leveraging cloud services for aspects of their critical software.



- Lesson 3: Lowering the transloading costs through techniques such as transloading hubs and containerization allows a much finer grained optimization of a global system.

In a world of low transloading costs, decisions no longer have to be based on the constraints of the global system. Instead, decisions can be optimized at the local subsystem without worrying about the impact on other subsystems.

We call the application of these three key lessons

(a) optimization through specialization,

(b) hybrid strategy maximizing economy of scale where possible while maintaining flexibility and agility where necessary and

(c) lowering transloading cost in the context of software architecture: localized optimization through selective specialization or LOtSS.

The rest of this article is about applying LOtSS in an enterprise scenario (Big Pharma) and an ISV scenario (LitwareHR)

LOtSS in the Enterprise – Big Pharma

How does LOtSS apply to the world of an enterprise? Let's examine it through the example of Big Pharma, a fictitious pharmaceutical company.

Big Pharma believes it does two things better than its competition: clinical trials and molecular research. However, it found that 80 percent of its IT budget is allocated to less strategic assets such as e-mail, CRM, and ERP. Of course, these capabilities are required to run the business, but none of them really help differentiate Big Pharma from its competitors. As Dr. Thomas (Big Pharma ficticious CEO) often says: "we have never beaten the competition because we had a better ERP system… it is all about our molecules."

Therefore, Dr. Thomas would be happy to get industry standard services for a better price. This is why Big Pharma is looking into leveraging "cloud offerings" to further optimize Big Pharma IT. The intent is to embrace a hybrid architecture, where economy of scale can be tapped for commodity services while retaining full control of the core capabilities. The challenge is to smoothly run and manage IT assets that span across the corporate boundaries. In LOtSS terms, Big Pharma is interested in optimizing its IT by selectively using the cloud where economy of scale is attainable. For this to work, Big Pharma needs to reduce to its minimum the "transloading" costs, which in this case means crossing the corporate firewall.

After analysis, Big Pharma is ready to push out of its data center two capabilities available at industry standard as a service by a vendor: CRM and e-mail. Notice that "industry average" doesn't necessarily mean "low quality." They are simply commoditized systems with features adequate for Big Pharma's needs. Because these providers offer these capabilities to a large number of customers, Big Pharma can tap into the economies of scale of these service providers, lowering TCO for each capability.

Big Pharma's ERP, on the other hand, was heavily customized and the requirements are such that no SaaS ISV offering matches the need. However, Big Pharma chooses to optimize this capability at a different level, by hosting it outside its data center. It still owns the ERP software itself, but now operations, hardware, A/C, and power are all the responsibility of the hoster. (See Figure 1.)

- Lesson 4: Optimization can happen at different levels. Companies can selectively outsource entire capabilities to highly specialized domain-specific vendors (CRM and e-mail in this example) or just aspects of an application (e.g. the operations and hardware as in the ERP).

The consequences of moving these systems outside of Big Pharma corporate boundaries cannot be ignored. Three common aspects of IT — security, management, and integration across these boundarie s— could potentially introduce unacceptable transloading costs.

From an access control perspective, Big Pharma does not want to keep separate username/passwords on each hosted service but wants to retain a centralized control of authorization rules for its employees by leveraging the already-existing hierarchy of roles in its Active Directory. In addition, Big Pharma wants a single sign-on experience for all its employees, regardless of who provides the service or where it is hosted. (See Figure 2.)

Figure 2: Single sign-on through identity federation and a Cloud STS.

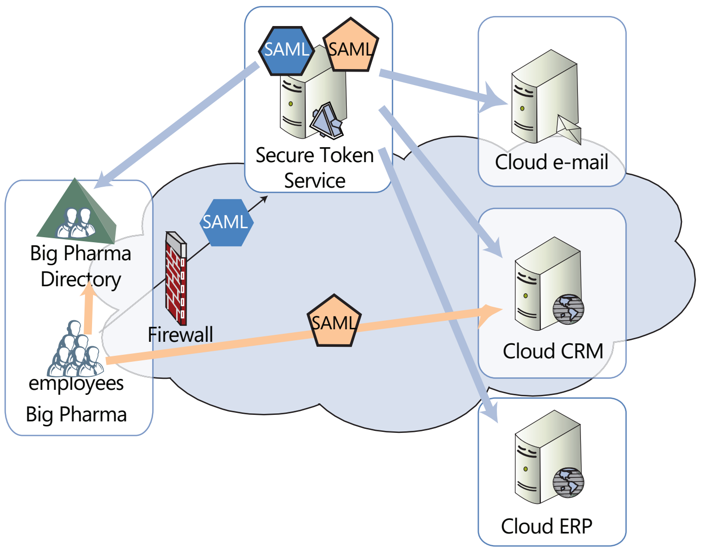
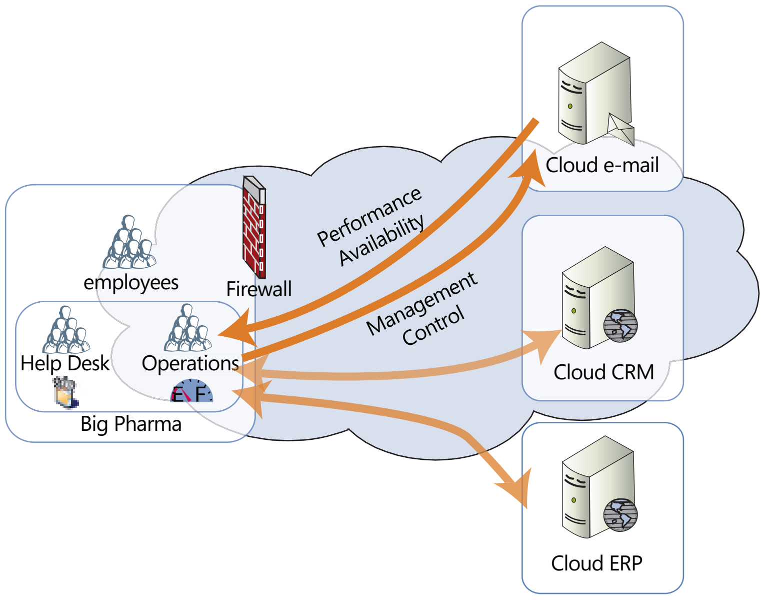


Figure 3: Big Pharma monitoring and controlling cloud services through management APIs.



high-end workstations with highly specialized software. Big Pharma chooses to run this software on-premises; due to its very complex visualizations requirements and high interactivity with its scientists, it would be next to impossible to run that part of the software in the cloud. However, simulations of these abstract models are tasks that demand highly intensive computational resources. Not only are these demands high, but they are also very variable. Big Pharma's scientists might come up with a model that requires the computational equivalent of two thousands machine/day and then nothing for a couple of weeks while the results are analyzed. Big Pharma could decide to invest at peak capacity but, because of the highly elastic demands, it would just acquire lots of CPUs that would, on average have a very low utilization rate. It could decide to invest at median capacity, which would not be very helpful for peak requests. Finally, Big Pharma chooses to subscribe to a service for raw computing resources. Allocation and configuration of these machines is done on-demand by Big Pharma's modeling software, and the provider they use guarantees state of the art hardware and networking. Because each simulation generates an incredible amount of information, it also subscribes to a storage service that will handle this unpredictably large amount of data at a much better cost than an internal high-end storage system. Part of the raw data generated by the cluster is then uploaded as needed into Big Pharma's on-premises systems for analysis and feedback into the modeling tools.

This hybrid approach offers the best of both worlds: The simulations can run using all the computational resources needed while costs are kept under control as only what is used is billed. (See Figure 4.)

Here again, for this to be really attractive, transloading costs (in this example, the ability to bridge the cloud-based simulation platform and the in-house modeling tool) need to be managed; otherwise, all the benefits of the utility platform would be lost in the crossing.

The other critical system for Big Pharma is clinical trials. After modeling and simulation, test drugs are tried with actual patients for effectiveness and side effects. Patients that participate in these trials need

One proven solution for decreasing the cross-domain authentication and authorization of "transloading costs" is federated identity and claims-based authorization. There are well-known, standards-based technologies to efficiently implement this.

- Lesson 5: Security systems today are often a patchwork of multiple ad-hoc solutions. It is common for companies to have multiple identity management systems in place. Companies like Big Pharma will favor solutions that implement standards-based authentication and authorization systems because it decreases transloading costs.

From a systems management perspective, Big Pharma employs a dedicated IT staff to make sure everything is working on agreed SLAs. Also, employees experiencing issues with the CRM, ERP, or e-mail will still call Big Pharma's helpdesk to get them resolved. It is therefore important that the IT staff can manage the IT assets regardless of whether they are internal or external. Therefore, each hosted system has to provide and expose a management API that Big Pharma IT can integrate into its existing management tools. For example, a ticket opened by an employee at Big Pharma for a problem with e-mail will automatically open another one with the e-mail service provider. When the issue is closed there, a notification is sent to Big Pharma that will in turn close the issue with the employee. (See Figure 3.)

- Caveat: Standards around systems management, across organization boundaries, are emerging (e.g. WS-Management) but they are not fully mainstream yet. In other words, "containerization" has not happened completely, but it is happening.

Now let's examine the systems that Big Pharma wants to invest in the most: molecular research and clinical trials. Molecular research represents the core of Big Pharma business: successful drugs. Typical requirements of molecular research are modeling and simulation. Modeling requires

Figure 4: A Big Pharma scientist modeling molecules on the on-premises software and submitting a simulation job to an on-demand cloud compute cluster.

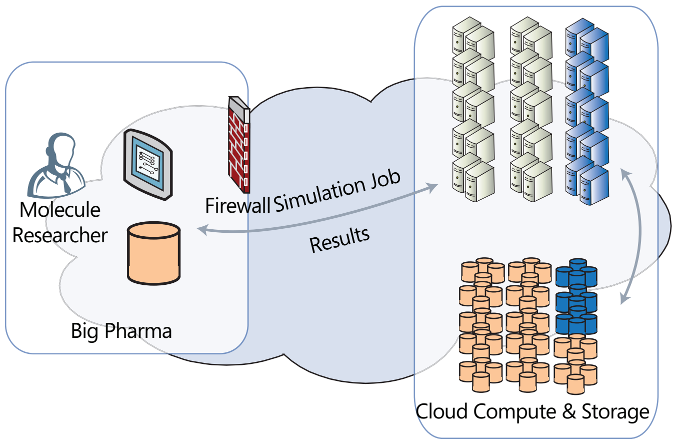
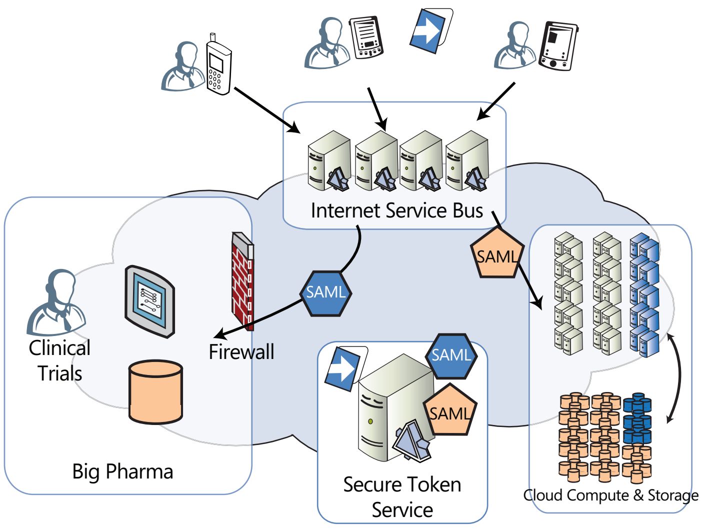


Figure 5: Big Pharma clinical trial patients submitting data to clinical trial system and simulation cluster.



to communicate back to Big Pharma all kinds of health indicators as well as various other pieces of information: what they are doing, where they are, how they feel, and so on. Patients submit the collected information to both the simulation cluster and the on-premises, specialized software that Big Pharma uses to track clinical trials.

Similar to the molecule modeling software, Big Pharma develops this system in-house, because of its highly specialized needs. One could argue that Big Pharma applied LOtSS many times already in the past. It is likely that the clinical trial software communication subsystem allowing patients to share their trial results evolved from an automated fax system several years ago to a self-service Web portal nowadays, allowing the patients to submit their data directly.

Participating patients use a wide variety of devices to interact with Big Pharma: their phones, the Web, their personal health monitoring systems. Therefore, Big Pharma decides to leverage a highly specialized cloud-based messaging system to offer reliable, secure communications with high SLAs. Building this on its own would be costly for the company. Big Pharma doesn't have the expertise required to develop and operate it, and they would not benefit from the economies of scale that the cloud service benefits from.

The Internet Service Bus allows Big Pharma to interact with patients using very sophisticated patterns. For example, it can broadcast updates to the software running on its devices to selected groups of patients. If, for any reason, Big Pharma's clinical trial software becomes unavailable, the ISB will store pending messages and forward them as the system becomes available again. (See Figure 5.)

Patients, on the other hand, are not Big Pharma employees, and therefore, are not part of Big Pharma's directory. Big Pharma uses federation to integrate patients' identities with its security systems. Patients use one of its existing Web identities such as Microsoft LiveID to authenticate themselves; the cloud identity service translates these authentication tokens into tokens that are understood by the services patients interact with.

In summary, this simplified scenario illustrates optimizations Big Pharma can achieve in its IT environment by selectively leveraging economy of scale prone specialized cloud services. This selection happens at different levels of abstraction, from finished services (e-mail, CRM, ERP) to building block services that only happen in the context of another capability (cloud compute, cloud identity services, cloud storage).

Applying LOtSS to LitwareHR

As described in the previous scenario, optimization can happen at many levels of abstraction. After all, it's about extracting a component from a larger system and replacing it with a cheaper substitute, while making sure that the substitution does not introduce high transloading cost (defeating the benefit of the substitution).

The fact is that ISVs have been doing this for a long time. Very few ISVs develop their own relational database nowadays. This is because acquiring a commercial, packaged RDBMS is cheaper, and building the equivalent functionality brings marginal value and cannot be justified.

Adopting commercial RDBMS allowed ISVs to optimize their investments, so there is no need to develop and maintain the "plumbing" required to store, retrieve, and query data. ISVs could then focus those resources on higher value components of their applications. But it does not come for free: There is legacy code, skills that need to be acquired, new programming paradigms to be learned, and so on. These are all, of course, examples of "transloading" costs that result as a consequence of adopting a specialized component.

The emergence and adoption of standards such as ANSI SQL and common relational models are the equivalent of "containerization" and have contributed to a decrease in these costs.

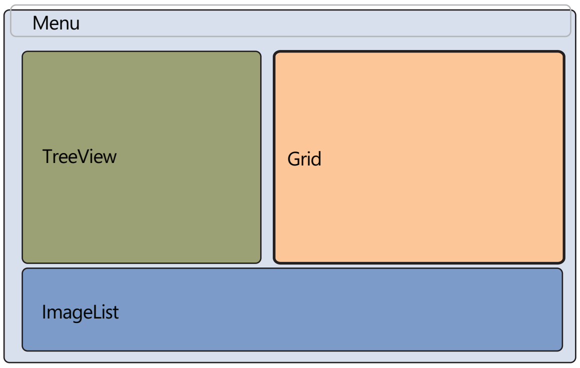
The market of visual components (e.g. Active-X controls) is just another example of LOtSS — external suppliers creating specialized components that resulted in optimizations in the larger solution. "Transloading costs" were decreased by de facto standards that everybody complied with tools. (See Figure 6.)

Cloud services offer yet another opportunity for ISVs to optimize aspects of their applications.

Cloud Services at the UI

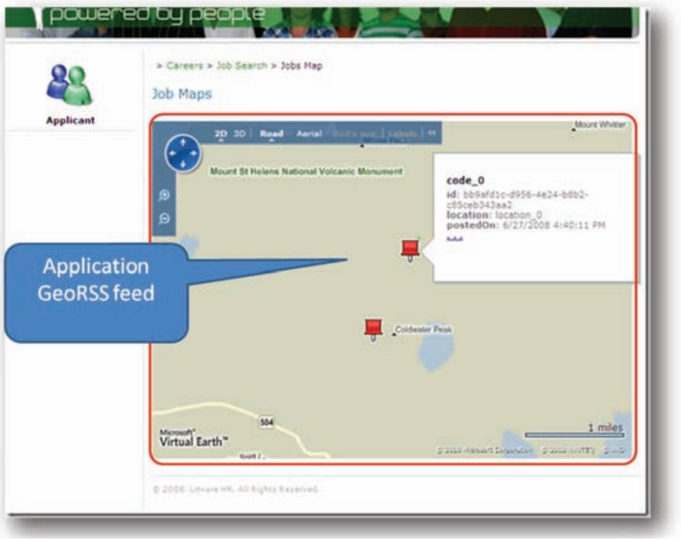
"Mashups" are one of the very first examples of leveraging cloud services. Cartographic services such as Microsoft Virtual Earth or

Figure 6: A window built upon three specialized visual parts, provided by three different vendors.



Google Maps allow anybody to add complex geographical and mapping features to an application with very low cost. Developing an equivalent in-house system would be prohibitive for most smaller ISVs, which simply could not afford the cost of image acquisition and rendering. Established standards such as HTML, HTTP, JavaScript, and XML, together with emerging ones like GeoRSS, contribute to lower the barrier of entry to these technologies. (See Figure 7.)

Figure 7: A Web mashup using Microsoft Virtual Earth, ASP.NET, and GeoRSS feeds.



Cloud Services at the Data Layer

Storage is a fundamental aspect of every nontrivial application. There are mainly two options today for storing information: a filesystem and a relational database. The former is appropriate for large datasets, unstructured information, documents, or proprietary file formats. The latter is normally used for managing structured information using well-defined data models. There are already hybrids even within these models. Many applications use XML as the format to encode data and then store these datasets in the filesystem. This approach provides database-like capabilities, such as retrieval and search without the cost of a fully fledged RDBMS that might require a separate server and extra maintenance.

The costs of running your own storage system include managing disk space and implementing resilience and fault tolerance.

Current cloud storage systems come in various flavors, too. Generally speaking there are two types: blob persistence services that are roughly equivalent to a filesystem and simple, semistructured entities persistence services that can be thought of as analogous to RDBMS. However, pushing these equivalences too much can be dangerous, because cloud storage systems are essentially different than local ones. To begin with, cloud storage apparently violates one of the most common system design principles: place data close to compute. Because everytime you interact with the store you necessarily make a request that goes across the network, applications that are too "chatty" with its store can be severely impacted by the unavoidable latency.

Fortunately, data comes in different types. Some pieces of information never or very seldom change (reference data, historical data, etc) whereas some is very volatile (the current value of a stock, etc). This differentiation creates an opportunity for optimization.

Once such optimization opportunity for leveraging cloud storage is the archival scenario. In this scenario, the application stores locally data that is frequently accessed or too volatile. All information that is seldom used but cannot be deleted can be pushed to a cloud storage service. (See Figure 8.)

With this optimization, the local store needs are reduced to the most active data, thus reducing disk capacity, server capacity, and maintenance.

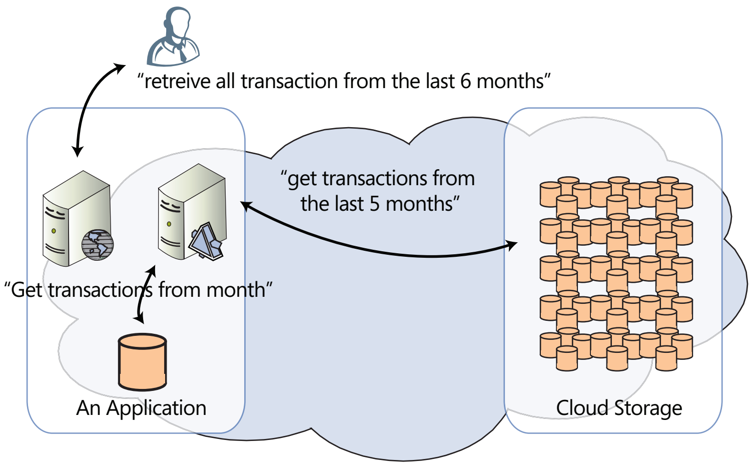
There are costs with this approach though. Most programming models of cloud storage systems are considerably different from traditional ones. This is one transloading cost that an application architect will need to weigh in the context of the solution. Most Cloud Storage services expose some kind of SOAP or REST interface with very basic operations. There are no such things as complex operators or query languages as of yet.

Authentication and Authorization in the Cloud

Traditional applications typically have either a user repository they own, often stored as rows in database tables, or rely on a corporate user directory such as Microsoft Active Directory.

Either of these very common ways of authenticating and authorizing users to a system have significant limitations in a hybrid in house — cloud scenario. The most obvious one is the proliferation of identity and permissions databases as companies continue to apply LOtSS. The likelihood of crossing organization boundaries increases dramatically and it is impractical and unrealistic that all these databases will be synchronized with ad-hoc mechanisms. Additionally, identity, authorization rules, and lifecycle (user provisioning and decommission) have to be integrated.

Figure 8: An application storing reference and historical data on a cloud storage service.

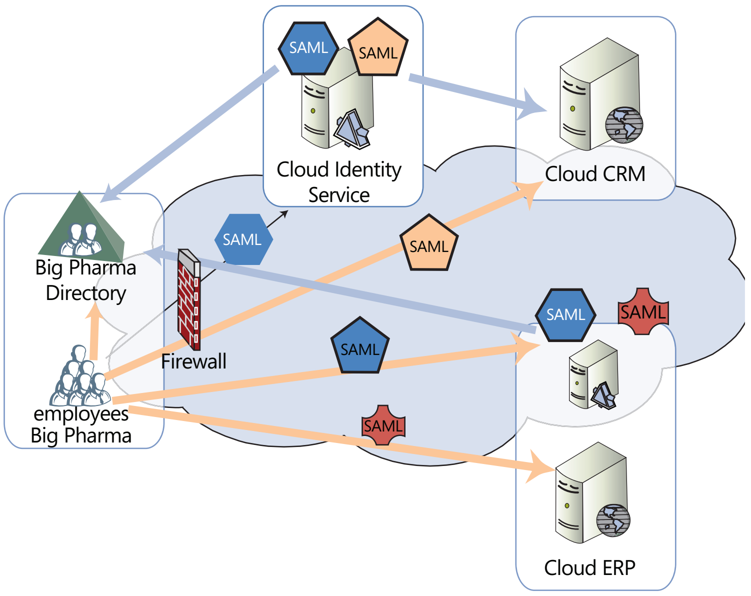

"retreive all transaction from the last 6 months"
"get transactions from the last 5 months"
"Get transactions from month"
An Application
Cloud Storage

Companies today have afforded dealing with multiple identities within their organization, because they have pushed that cost to their employees. They are the ones having to remember 50 passwords for 50 different, not integrated systems.

• Consider this: When a company fires an employee, access to all the systems is traditionally restricted by simply preventing him/her to enter the building and therefore to the corporate network. In a company leveraging a cloud service, managing its own username/password as opposed to federated identity, the (potentially annoyed and resentful) fired employee is theoretically capable of opening a browser at home and logging on to the system. What could happen then is left as an exercise to the reader.

As mentioned in the Big Pharma scenario, identity federation and claims-based authorization are two technologies that are both standards based (fostering wide adoption) and easier to implement today by platform-level frameworks built by major providers such as Microsoft. In addition to frameworks, ISVs can leverage the availability of cloud identity services, which enables ISVs to optimize their infrastructure by pushing all this infrastructure to a specialized provider. (See Figure 9.)

Figure 9: Big Pharma consuming two services. CRM leverages cloud identity for claim mapping and ERP uses a hosted based STS.



Decompose-Optimize-Recompose

The concept introduced in this article is fairly straightforward and can be summarized in the following three steps: Decompose a system in smaller subsystems; identify optimization opportunities at the subsystem level; and recompose while minimizing the transloading cost introduced by the new optimized subsystem. Unfortunately, in typical IT systems, there are too many variables to allow a magic formula for discovering the candidates for optimization. Experience gained from previous optimizations, along with tinkering, will likely be the best weapons to approach LOtSS. This said, there are high-level heuristics that can be used. For example, not all data is equal. Read-only, public, benefitting from geo distribution data such as blog entries, product catalogs, and videos is likely to be very cloud friendly; on the other hand, volatile, transaction requiring, regulated, personally identifying data will be more on-premises friendly. Similarly, not all computation is equal; constant predictable computation loads are less cloud attractive than variable, unpredictable loads, which are better suited for utility type underlying platform.

Conclusion

LOtSS uses the cloud as an optimization opportunity driving overall costs down. Although this "faster, cheaper, better" view of the cloud is very important and is likely to be a key driver of cloud utilization, it is important to realize that it is only a partial view of what the cloud can offer. The cloud also presents opportunities for "not previously possible" scenarios.

IT history has shown that over time, what was unique and strategic becomes commodity. The only thing that seems to stay constant is the need to continuously identify new optimization opportunities by selectively replacing subsystems with more cost-effective ones, while keeping the cost introduced by the substitute low. In other words, it seems that one of the few sustainable competitive advantages in IT is in fact the ability to master LOtSS.

Resources

"Data on the Outside vs. Data on the Inside, " by Pat Helland
http://msdn.microsoft.com/en-us/library/ms954587.aspx

Identity Management in the Cloud
http://msdn.microsoft.com/en-us/arcjournal/cc836390.aspx

Microsoft Identity Framework, codename "Zermatt"
https://connect.microsoft.com/site/sitehome.aspx?SiteID=642

About the Authors

Gianpaolo Carraro – Senior Director, Architecture Strategy. In his current role, Gianpaolo leads a team of architects driving thought leadership and architectural best practices in the area of Software + Services, SaaS, and cloud computing. Prior to Microsoft, Gianpaolo helped inflate and then burst the .com bubble as cofounder and chief architect of a SaaS startup. Gianpaolo started his career in research as a member of the technical staff at Bell Laboratories. You can learn more about him through his blog http://blogs.msdn.com/gianpaolo/.

Eugenio Pace – Sr Architect, Architecture Strategy – Eugenio is responsible for developing architecture guidance in the area of Software + Services, SaaS, and cloud computing. Before joining the Architecture Strategy group, he worked in the patterns & practices team at Microsoft, where he was responsible for delivering client-side architecture guidance, including Web clients, smart clients, and mobile clients. During that time, his team shipped the Composite UI Application Block, and three software factories for mobile and desktop smart clients and for Web development. Before joining Patterns and Practices, he was an architect at Microsoft Consulting Services. You can find his blog at http://blogs.msdn.com/eugeniop.

Architecture Journal Profile: Microsoft Robotics Group

While not widely considered in the enterprise except within certain verticals, such as manufacturing, robotics technology is today available to a broad software developer segment and is expected to be more and more present in the enterprise architectural landscape. With that in mind, we interviewed two thought leaders in the Microsoft Robotics Group about their careers and the future of robotics in the enterprise.

AJ: Who are you, and what do you do?
TT: I'm Tandy Trower. I'm the general manager of the Microsoft Robotics Group and founder of the Robotics initiative at Microsoft.
HFN: My name is Henrik Frystyk Nielsen. I am the group program manager in the Microsoft Robotics Group, and I'm very excited about bringing robotics into line-of-business (LOB) applications.

AJ: Robotics would seem to be a far remove from traditional LOB applications. Where do these domains meet?
TT: Let me answer that by describing the origins of the robotics initiative. The robotics community is quite diverse, from research on cutting-edge technologies (such as the Dynamic Source Routing protocol in the DARPA urban challenge) to very early segments of the commercial industries, as well as the existing industrial automation segment that's looking for new marketplaces. The robotics community, through their various representatives, asked Microsoft for software assets that would address such challenges as maintainability and distributed concurrent applications.

When I presented this proposal first to Bill Gates and Craig Mundie, Craig pointed out that he had been incubating a new piece of technology to deal with the challenges of loosely coupled asynchronous distributed software. Henrik was a part of the team that created these. Although Henrik's team did not have robotics in mind originally, the problem set that they were solving was very similar to the problem set the robotics community was facing.

These technologies, known today as CCR (Concurrency and Coordination Runtime) and DSS (Decentralized Software Services), form the foundation of the robotics toolkit. The core services of the robotics toolkit, to this day, have remained a general purpose programming application to deal with asynchronous applications. CCR is the piece that allows you to easily grapple with the challenges of writing an application on a multicore or multiprocessor system. DSS is that companion piece for scaling out across the network that allows you to have a consistent programming model locally and across the network and across heterogeneous types of devices.

As soon as we made the robotics toolkit available to the marketplace, customers outside of the robotics space started taking a look and kind of lifted up the hood and said "That engine is good for my kind of applications as well. I have the same kind of problem."
HFN: We recognized early on that more and more applications today are connections of loosely coupled components, and the challenge is about orchestration, about services. We wanted to bring a high-level application model to bear that inherently knows how to stitch together loosely coupled components. Bad things can happen—each component can fail at any point in time—but you want the application to survive, you don't want it to just roll over and die. And at the same time we had to deal with and harness concurrency for better scale, better responsiveness, and applications that don't fail.

Some customers recognized their own problem space based on the description of the basic components. For example, financial trading systems people said, "This is exactly about lots of data coming in, figuring out when to trade, and when not to, with independent agencies looking at the data with different viewpoints and decision making process flows"—a complex application that turns out to be similar to a robotics application where your information comes from the wheels, the lasers, and whatever else, and you have to decide, should I stop or go.

Other customers came from the robotics side. Robotics in fact is a great enabler. We had hobbyist roboticists come to us saying, "I've played with this toolkit over the weekend to build a robot, then I came into work and realized that I could apply the same matrices that I had done over the weekend to solve the problem in the enterprise."
TT: Until now, we've bundled these assets into the robotics toolkit, but increasingly, customers outside the robotics space are saying, "These are very valuable assets, why do we have to buy a robotics toolkit to get them?" So we will be addressing that by offering these components in a more palatable way to those customers; rather than having to buy them under the robotics cover, they'll be able to get those assets directly.

AJ: What advice, not necessarily related to robotics, would you share with aspiring architects?
TT: Look for where things are going, keep an eye open to new opportunities. Certainly, at Microsoft there are leadership opportunities and opportunities for different kinds of leaders. Some leaders, like myself, are better in startup types of activities; others are better at maintaining stable organizations that are mature.
HFN: When you get to an expert level of understanding in a certain area, there's a tendency to dive into that area and forget some of the other

areas around you. You naturally build cultural walls between your area of knowledge and what you think other areas are doing. The experience of our team certainly demonstrates the benefit of breaking down those walls, being able to step back and recognize there's a community, that there's very great overlap we better talk with each other. It's a huge advantage and I think it's something that characterizes a great leader.

AJ: The architect role requires understanding technical trends to get the best business benefit. How do you stay up to date?
TT: The Web provides almost unlimited ways for connecting—blogs, forums, feeds, virtual libraries, wikis. Attending related conferences is another good way. A lot of the people in this industry are very heads down and focused. I think part of staying current is simply taking the time to look up and notice what else is going on. You might find some synergies there that you never even considered.
HFN: Just 10, 15 years ago, it was about getting information; now the challenge has become filtering the information. When there's so much information that you can't possibly consume it, you have to rely on new mechanisms. Figuring out how to get high-quality information about the things that actually solve problems is a critical task. Blogs, social networks, and newsfeeds of all kinds are a start.

AJ: Name the most important person you have ever met in this industry. What made him/her so important?
TT: There are so many. Of course, Bill Gates—he's the reason I've been at Microsoft for as long as I have. Also our more direct sponsor, Craig Mundie. Craig has been my manager a number of different times and is one of the forward-looking executives at Microsoft, having the ability to see where things are going, and then allowing me to participate, through what he was doing, and actually innovate in some of those areas.
HFN: I would add from my perspective Tim Berners-Lee, inventor of the Web, whose story shows how a single person effectively changed how the world operates today, from how we to work to how we talk.

AJ: If you could turn the clock back on your career, would you do anything differently?
HFN: Obviously, you think of many of your choices as discrete decisions, so it's funny how very planned and deliberate your career appears in hindsight. I'm fortunate to have had many opportunities come my way. If you're in a position where many opportunities arise, chances are good that an avenue will be open for you when you are ready for the challenge. But if you are in a situation where nothing really comes by, then you might be ready for a long time before an opportunity materializes.
TT: When I think through all of my life experiences and career experiences, I've always had a chance to kind of reinvent myself; it has been very satisfying to work in new areas. If anything, I would advise anyone to just be more aware of the fact that things never remain the same, that the industry evolves, so don't to get too stuck on anything because the landscape is going to change. Microsoft has had to work very hard to maintain its edge, and it only does that by continuing to refresh itself and to look to new things. So, stay fluid, and roll with the punches.

AJ: What does the future look like? What do you hope to accomplish in the next few years?
TT: Well, the robotics toolkit is part of a fairly ambitious goal to create a platform that would allow a new industry to emerge. Robotics has traditionally been in the industrial sector; a new business is now emerging from the personal side of robotics. Personal robotics has not arrived yet, but in many ways, it maps to the early PC industry of the late 1970s: When PCs first entered the marketplace, they kind of looked like toys. There wasn't a clear understanding of what they were useful for.

Ultimately, I see robotics everywhere, like PCs, a very pervasive and ubiquitous technology. That's going to take time, but I do believe that we will start to see that happen in three to five years.
HFN: I agree. Robotics has been on the edge for a long time, but it's getting close to taking off in dramatic ways. It's is going to drive how we interact with information and how we communicate, in social settings and industrial settings.
TT: One marvel of human design is that we are built from very simple processors that are massively connected together. The present computing paradigm is as if we've created a brain out of a single neuron. We've done wonderful things with it, but now we have the potential to define a technology that's analogous to multiple neurons interacting together in a network effect. That's reflected also in our social networks, ecological networks, or whatever you want to talk about that's in some sense living.

Robotics represents the paradigm shift in the evolution of PC technology. The future is not going to be a single core processor sitting on your desk. You can now use technologies like CCR and DSS to develop a new generation of applications. It's going to open up some significant horizons: Robotics is just the starting point.

Henrik Frystyk Nielson

Henrik Frystyk Nielsen is best known for his contributions to the World Wide Web infrastructure. Notably, he is a co-author of the foundation protocol for the Web, Hypertext Transfer Protocol (HTTP) and a co-author of SOAP. Together with George Chrysanthakopoulos, he designed the application model behind Microsoft Robotics Developer Studio and authored the associated protocol specification Decentralized Software Services Protocol (DSSP).

Before joining Microsoft, Henrik worked as technical staff at the World Wide Web Consortium (W3C) hosted by Massachusetts Institute of Technology. Henrik received his master's in electrical engineering from Aalborg University, completing his thesis at CERN under World Wide Web inventor Sir Tim Berners-Lee.

Trandy Trower

Tandy Trower has a 27-year history in starting up new products and technology initiatives at Microsoft, bringing to market new products as diverse as Microsoft Flight Simulator to Microsoft Windows. In addition, as an early proponent of the importance of design in human-computer interaction he establishing the company's early user-centered design community, founding the first usability labs and product design roles. He continues to drive strategic new technology directions for the company and incubating new projects. He leads the new Microsoft robotics initiative which is delivering a software platform and tools as a catalyst for emerging personal robotics market.

On Distributed Embedded Systems

by Arvindra Sehmi

Summary
Thinking of distributed embedded systems (DES)—let alone the more general area of embedded computing—as a unified topic is difficult. Nevertheless, it is a vastly important topic and potentially represents a revolution in information technology (IT). DES is driven by the increasing capabilities and ever-declining costs of computing and communications devices, resulting in networked systems of embedded computers whose functional components are nearly invisible to end users. Systems have the potential to radically alter the way people interact with their environment by linking a range of devices and sensors that will allow information to be collected, shared, and processed in unprecedented ways.

DES conjures up images of pervasive collaboration among connected devices, such as tiny, stand-alone, embedded microcontrollers, networking devices, embedded PCs, robotics systems, computer peripherals, wireless data systems, sensors, and signal processors. Combining information from in-place sensors with information from other sources on the network will enable new, rich scenarios to be realized in automotive and avionics control systems, local environmental monitoring for precision agriculture, medical systems, personal health-care monitoring, and manufacturing and process automation, for example.

Widespread use of DES throughout society could dwarf previous revolutions in IT for two reasons. Firstly, Moore's law is primarily driving device miniaturization and reduced power consumption (rather than increased speed). Secondly, the industry is fast becoming better equipped with development tool chains (software included) and pervasive standards-based communications protocols. As a consequence, embedded computing and communications technology can quite literally be everywhere and extend into all aspects of life—an invisible component of almost everything in everyone's surroundings.

Contrasting DES with Traditional Distributed Systems

DES differs from traditional distributed systems in important ways. Embedded systems will eventually interconnect millions, maybe billions, of nodes. This will require changes in the way nodes interact with one another. Broadcast, not peer-to-peer, communications will be the norm. The sheer size of connected clusters of applications will necessitate the use of statistically correct rather than deterministic algorithms for resource accounting, fault detection and correction, and system management. These clusters will merge and dissolve rapidly to host functionality that is of interest to groups formed specifically for that purpose. Doing this successfully requires new approaches to naming, routing, security, privacy, resource management, and synchronization. Heterogeneity will factor greatly in the design, implementation, and operation of DES, as will cross-cutting concerns such as dependability, energy-aware computing, critical systems engineering, security, real-time programming, and resource-bounded computing. Some of these factors are familiar to traditional application developers, but many are simply viewed as esoteric, boundary issues.

DES tends to be tightly coupled to the physical world and will often be invisible when things are working properly. In contrast to desktop computers, which are part of the office or home furniture, DES instead will be integrated into the furniture and other objects in the environment. You will interact with these objects and devices, but will be unlikely to think of them, as you do when interacting with a computer.

DES components are highly resource constrained. They are typically small, wirelessly connected, bandwidth limited, and operating under physical constraints such as limited energy and the need for adequate heat dissipation. Since they would be integrated in buildings, bridges, vehicles, and so on, they would be expected to last as long as the objects in which they are embedded. The expectation of longevity needs to be taken into account when designing, deploying, and managing these systems. Heterogeneity will be the norm because of the large number of interacting elements that make up DES, so interoperability is a key concern. Managing these constraints and creating a system that functions properly while remaining understandable and manageable by human operators, users, and casual passersby, is a great challenge for DES designers—arguably much more of a challenge than that posed by traditional distributed systems design. The DES scenarios described later in this paper should clarify these points.

Market Opportunity

DES is a fast-growth area of the computing industry and is indicative of the long-term trend of moving away from centralized, high cost, low volume products toward distributed, low cost, high volume products.

The next step in this process is the emergence of massively distributed systems, that is, distributed embedded systems that are connected to enterprise systems and the Internet (or the so-called, Cloud). These systems will penetrate even more deeply into the fabric of society and become the information power grids of the 21st century. As noted, they will be ubiquitous, most will operate outside

Table 1: Estimated total value, through 2006, and growth, through 2010, of major industry sectors using embedded systems (based on the FAST report [2] and others).

ES = embedded systems; EC = electronic components

Industry Sector	Annual Global Value	ES Value (%)	ES Value	EC Growth (%)	ES Growth (%)
Automotive	800 b€	40%	320 b€	10%	10%
Avionics/Aerospace	750 b€	50%	370 b€	5%	14%
Industrial Automation	200 b€	55%	110 b€	5%	7%
Telecommunications	1000 b€	55%	550 b€	9%	15%
Consumer Electronics and Intelligent Homes	300 b€	60%	180 b€	8%	15%
Health & Medical Systems	130 b€	40%	50 b€	?	18%
Total	3180 b€		1580 b€		

the normal cognizance of the people they serve, and most will be based on embedded systems that present nontraditional computing interfaces to their users. Their full potential will see them engineered to operate as distributed utilities, much like the energy, water, transportation, and media broadcast businesses do today [1]. The major industry sectors where DES is likely to be used are automotive, avionics/aerospace, industrial automation (and robotics); telecommunications; consumer electronics and intelligent homes; health and medical systems. The market opportunity of these sectors is significant and the reader is referred to the FAST report [2] for more details. In summary though and shown in Table 1, the overall value of the embedded sector worldwide is about 1600 billion € per year; the three largest markets for embedded systems are telecommunications, automotive, and avionics/aerospace with combined value worldwide of 1,240 billion € per year; and these three largest markets are characterized by growth rates in excess of 10 percent.

The nature and characteristics of distributed embedded systems are explored below through some example scenarios drawn from the industry sectors mentioned above. This will help draw similarities to traditional distributed systems development while also recognizing that there are unique systems-design implications, implementation issues, and approaches to solution development.

Scenarios

Automotive Telematics Scenario

In the automotive and avionics industry, embedded systems provide the capability of reaching new levels of safety and sustainability that otherwise would not be feasible, while adding functionality, improving comfort, and increasing efficiency. Examples of this include improved manufacturing techniques, driver-assistance systems in cars that help prevent accidents, and advanced power-train management concepts that reduce fuel consumption and emissions.

In Western Europe, the “100 percent safe” car is envisioned. It will have sensors, actuators, and smart embedded software, ensuring that neither the driver nor the vehicle is the cause of any accident. This concept extends to all aspects of the driving experience: in-car entertainment, information services, and car-to-car and car-to-infrastructure communication.

For example, the car would know who is allowed to drive it and who is driving it; where it is; where it is going and the best route to its destination; and it would be able to exchange information with vehicles around it and with the highway. It would monitor its own state (fuel levels, tires, oil pressure, passenger compartment temperature and humidity, component malfunction, need for maintenance) and the state of the driver (fatigue, intoxication, anger). The car would first advise and then override the driver in safety-critical situations, use intelligent systems to minimize fuel consumption and emissions, and contain an advanced on-board entertainment system.

Radio-frequency identification (RFID) smart tags within the major car components would communicate with other components during manufacture to optimize the process, communicate with in-car systems during the car’s working life to optimize maintenance cycles, and enable environmentally friendly disposal of the car and its components at the end of its life.

Challenges and Issues: To enable this scenario, components would need to be embedded in long-lived physical structures (such as bridges, traffic lights, individual cars, and perhaps even the paint on the roads). Some components will be permanently connected to a network, but many would be resource constrained (e.g., in terms of power) while computing data and thus communicating it wirelessly only when necessary. The many pieces of such a system will of necessity be heterogeneous, not only in form but also in function. There may be subsystems that communicate to consumers in private vehicles, others that relay information from emergency vehicles to synchronize traffic lights, still others that provide traffic data and analysis to highway engineers, and perhaps some that communicate to law enforcement.

How information will be communicated to those interacting with the system is of great importance in such an environment. Safety is a critical concern, and issues of privacy and security arise as well, along with concerns about reliability.

Precision Agriculture Scenario

Incorporating DES technology into agriculture is a logical development of the advances in crop management over the last few decades. Despite deep understanding and knowledge on the part of farmers about how to adjust fertilizers, water supplies, and pesticides, and so on, to best manage crops and increase yields, a multitude of variations still exist in soil, land elevation, light exposure, and microclimates that make general solutions less than optimal, especially for highly sensitive crops like wine grapes and citrus fruit.

The latest developments in precision agriculture deploy fine-grained sensing and automated actuation to keep water, fertilizer, and pesticides to a minimum for a particular local area, resulting in better yields, lower costs, and less pollution-causing runoff and emissions. Furthermore, the data collected can be analyzed and incorporated as feedback control to adjust irrigation flow rate and duration tuned to local soil conditions and temperature. Sensors that can monitor the crop itself (for example, sugar levels in grapes) to provide location-specific data could prove very effective.

In the future DES might be used to deploy sensors for the early detection of bacterial development in crops or viral contamination in livestock, or monitor flows of contaminants from neighboring areas and send alerts when necessary. In livestock management, feed and vitamins for individual animals will be adjusted by analyzing data from networks of ingestible sensors that monitor amounts of food eaten, activity and exercise, and health information about individual animals and the state of the herd as a whole.

Challenges and Issues: In this scenario, embedded components must be adaptive, multimodal, and able to learn over time. They will need to work under a wide range of unpredictable environmental conditions, as well as to interact with fixed and mobile infrastructure and new elements of the system as they are added and removed at varying rates of change.

Aviation and Avionics Scenario

The European Commission has set goals for the aviation industry of reducing fuel consumption by 30 percent by 2021 through the use of embedded systems. This may be a high goal to be achieved solely through the use of technology. But in this industry, the goals appear to very ambitious across the board.

The unmanned aerial vehicle (UAV) for use in surveillance and in hazardous situations such as fire fighting promises to be cheaper, safer, and more energy efficient to operate than conventional aircraft. There are apparently many different kinds of UAVs under development: some with long-duration operational cycles and extensive sensor suites; some with military defense and attack capability; others that are small enough to be carried and deployed by individuals; and, in the future, tiny, insect-like, UAVs providing a flying sensor network.

The aircraft of the future will have advanced networks for on-board communication, mission control, and distributed coordination between aircraft. These networks will support advanced diagnosis, predictive maintenance, and in-flight communications for passengers. For external communication, future aircraft will communicate with each other in spontaneous, specific-for the-purpose ways similar to peer-to-peer networks.

Challenges and Issues: The aviation and avionics industry has specific needs in terms of security, dependability, fault tolerance and timeliness, stretching the limits of distributed embedded systems design and implementation. The whole system, if not each of its embedded components, needs to be high precision, predictable, and robust for 100 percent operational availability and reliability. It must enable high bandwidth, secure, seamless connectivity of the aircraft with its in-flight and on-ground environment. It should support advanced diagnosis and predictive maintenance to ensure a 20- to 30-year operational life span.

DES design environments and tools will need to provide significant improvements in product development cycles, ongoing customizations, and upgrades beyond those achievable with current distributed systems development tools. Design advances in fast prototyping, constructive system composition, and verification and validation strategies will be required to manage this complexity.

Manufacturing and Process Automation Scenario

Embedded systems are important to manufacturing in terms of safety, efficiency, and productivity. They will precisely control process parameters, thus reducing the total cost of manufacture. Potential benefits from integrating embedded control and monitoring systems into the production line include: better product quality and less waste through close process control and real-time quality assurance; more flexible, quickly configured production lines as a result of programmable subsystems; system health monitoring, which leads to more-effective, preventive and lower-cost maintenance; safer working environments due to better monitoring and control; and better component assembly techniques, such as through the use of smart RFID tags

Challenges and Issues: There are many implications of this industry scenario for DES. One is a need for better man-machine interactions in what is fundamentally a real-time, man-plus-machine control loop. Providing better interactions will improve quality and productivity by ensuring that there are no operator errors, as well as by reducing accidents. Availability, reliability, and continuous quality of service are essential requirements for industrial systems achieved through advanced control, redundancy, intelligent alarming, self-diagnosis, and repair. Other important issues are the need for robustness and testing, coherent system-design methodology, finding a balance between openness and security, integrating old and new hardware with heterogeneous systems, and managing obsolescence.

Medical and Health-care Scenario

Society is facing the challenge of delivering good-quality, cost-effective health care to all citizens. Medical care for an aging population, the cost of managing chronic diseases, and the increasing demand for best-quality health care are major factors in explaining why health-care expenditures in Europe are already significant (8.5 percent of GDP) and rising faster than overall economic growth. Medical diagnosis and treatment systems already rely heavily on advances in embedded systems. New solutions that mix embedded intelligence and body-sensing techniques are currently being developed [3], and current advances in this area address patient-care issues such as biomedical imaging, remote monitoring, automatic drug dispensing, and automated support for diagnosis and surgical intervention.

Challenges and Issues: The medical domain represents a complex and diverse arena for extraordinary developments that deploy a wide range of embedded systems. Implanted devices such as pacemakers and drug dispensers are commonplace, but need to become more sophisticated, miniaturized, and connected to networks of information systems. Wearable devices for monitoring and managing cholesterol, blood sugar, blood pressure, and heart rate must be remotely connected to the laboratory and to the operating room in a secure and reliable manner from beginning to end. Robotic devices are being used today to guide and perform invasive surgery requiring high-integrity engineering practices not even imagined in a typical "mission-critical" enterprise application.

Mobility Scenario

Combining mobile communications with mobile computing is allowing people to talk to others and access information and entertainment anywhere at any time. This requires ubiquitous, secure, instant, wireless connectivity, convergence of functions, global and short-range sensor networks and light, convenient, high-functionality terminals with sophisticated energy management techniques. Such environments will enable new forms of working with increased productivity by making information instantly available, when needed in the home, cars, trains, aeroplanes and wider-area networks. Imagine a hand-held or wearable device giving easy access to a range of services able to connect via a range of technologies including GSM, GPS, wireless, Bluetooth and via direct connection to a range of fixed infrastructure terminals. Potential applications and services include: Entertainment, education, internet, local information, payments, telephony, news alerts, VPNs, interfaces to medical sensors and medical services, travel passes and many more.

Challenges and Issues: Devices would need to reconfigure themselves autonomously depending on patterns of use and the available supporting capabilities in environment or infrastructure and be able to download new services as they became available. To develop such infrastructure, the gap between large, enterprise systems and embedded components would need to be bridged. Significant developments are required in technology for low-power and high performance computing, networked operating systems, development and programming environments, energy management, networking and security.

Issues which need to be resolved in the infrastructure to support these kinds of scenarios include the provision of end-to-end ubiquitous, interoperable, secure, instant, wireless connectivity to services. Simultaneously the infrastructure must allow unhindered convergence of functions and of sensor networks. Addressing the constraints imposed by power management (energy storage, utilization and generation) at the level of the infrastructure and mobile device poses a major challenge.

Home Automation and Smart Personal Spaces Scenario

By deploying DES in the home, an autonomous, integrated, home environment that is highly customizable to the requirements of individuals can be foreseen. Typical applications and services available today include intruder detection, security, and environmental control. But in the future, applications and services to support the young, elderly, and infirm will be developed, and these may have the ability to recognize individuals and adapt to their evolving requirements, thereby enhancing their safety, security, and comfort. By tying in with applications and services described in the medical/healthcare and mobility scenarios, smart personal spaces could be developed.

Challenges and Issues: Multi-disciplinary, multi-objective design techniques that offer appropriate price and performance, power consumption, and control will have to be used if we are to realize the potential of embedded systems for home entertainment, monitoring, energy efficiency, security, and control. Such systems will require significant computational, communication, and data-storage capabilities. The mix of physical monitoring and data-based decision support by some form of distributed intelligence will rely on the existence of seamlessly connected embedded systems and the integration of sensors and actuators into intelligent environments. These systems will be characterized by ubiquitous sensors and actuators and a high-bandwidth connection to the rest of the world. Technologies will need to be developed that support sensing, tracking, ergonomics, ease-of-use, security, comfort, and multimodal interaction.

Key to achieving this result will be developing wireless and wired communications and techniques for managing sensor information, including data fusion and sensor overloading. The challenges are to make such systems intelligent, trustworthy, self-installing, self-maintaining, self-repairing, and affordable, and to manage the complexity of system behavior in the context of a large number of interoperable, connected, heterogeneous devices. These systems will need to operate for years without service, be able to recover from failure, and be able to supervise themselves.

Managing these embedded systems will require support of all aspects of the life cycle of the application and service infrastructures, including ownership, long-term storage, logging of system data, maintenance, alarms, and actions by the provider (emergency, medical, or security) services, authorization of access and usage, and charging and billing under a range of different conditions of use.

Technical Imperatives

Some of the most challenging problems facing the embedded systems community are those associated with producing software for real-time and embedded systems. Such systems have historically been targeted to relatively small-scale and stand-alone systems, but the trend is toward significantly increased functionality, complexity, and scalability, as real-time embedded systems are increasingly being connected via wireless and wired networks to create large-scale, distributed, real-time, and embedded systems. These combined factors [4] require major technical imperatives to be addressed by both industry and research establishments: Self-configuration and adaptive coordination; (b) Trustworthiness; (c) Computational models; and (d) Enabling technologies. Let's now look at each technical imperative in more detail.

Self-configuration and Adaptive Coordination

Self-configuration is the process of interconnecting available elements into an ensemble that will perform the required functions at the desired performance level. Self-configuration in existing systems is realized through the concepts of service discovery, interfaces, and interoperability. But embedded systems, which appear in hybrid environments of mobile and static networks with nodes of varying capability, energy availability, and quality of connectivity, are plagued by diverse and energy-limited wireless connectivity—making low power discovery a challenge. Also scalable discovery protocols, security, and the development of adequate failure models for automatically configured networks will require that solutions be developed.

Adaptive coordination involves changes in the behavior of a system as it responds to changes in the environment or system resources. Coordination will not necessarily be mediated by humans because DES could be so large and the time scale over which adaptation needs to take place too short for humans to intervene effectively. Achieving adaptive coordination in DES will draw on the lessons learned from adaptive coordination in existing distributed systems, but it will also require meeting the radical new challenges posed by the physically embedded nature of collaborative control tasks and large numbers of nodes, and further complicated by the relatively constrained capabilities of individual elements. Thus, to achieve adaptability in DES, solutions are needed in decentralized control and collaborative processing, and techniques must be developed to exploit massive redundancy to achieve system robustness and longevity.

Trustworthiness

If we can expect DES to be deployed in large numbers and become an essential part of the fabric of everyday life, five technical imperatives much be taken into account in their design from the outset: reliability, safety, security, privacy, and usability.

On reliability, current monitoring and performance-checking facilities, and verification techniques are not easily applicable to DES because of their large number of elements, highly distributed nature, and environmental dynamics.

In terms of safety or the ability to operate without causing accidents or loss, bounded rational behaviors are essential, especially in the face of real-time systems and massive DES likely to exhibit emergent or unintended behaviors.

It may be virtually impossible to distinguish physical from system boundaries in a large-scale DES, making security a big challenge. And, considering that networking of embedded devices will greatly increase the number of possible failure points, security analysis may prove even more difficult.

Privacy and confidentiality policies will be exacerbated by the pervasiveness and interconnectedness of DES. Users will be monitored, and vast amounts of personal information will be collected. Thus, implementing privacy policies, such as acquiring consent in a meaningful fashion in large scale networks, will be very difficult.

Related to all of the above, embedded systems need to be usable by individuals with little or no formal training. Unfortunately, usability and safety often conflict, so trade-offs will need to be made. Understanding the mental models people use of the systems with which they interact is a good way for designers to start addressing issues of embedded systems' usability and manageability.

Computational Models

New models of computation are needed to describe, understand, construct, and reason about DES effectively. Understanding how large aggregates of nodes can be programmed to carry out their tasks in a distributed and adaptive manner is a critical research area. Current distributed computing models, such as distributed objects and distributed shared memory, do not fully address all of the new requirements of DES. Furthermore, DES's tight coupling to the physical world, the heterogeneity of their systems, the multitude of elements, plus timing and resource constraints, among other things, demonstrate the need for a much richer computing model. Such a model will need to incorporate resource constraints, failures (individual components may fail by shutting down to conserve energy, for example), new data models, trust, concurrency, and location.

For example, in a sensor network, subsystems from different vendors should interoperate easily and be integrated seamlessly into the rest of the IT infrastructure, providing intuitive interfaces for remote and novice users. Multiple, concurrent clients will be exercising different functionalities of the system for different purposes, and although resources may not be the most stringent constraint for the system, it has to be self-monitoring and aware of its resources. It must have a certain level of autonomy to decide on the best use of available resources to fulfill multiple users' concurrent and uncoordinated requests. The complexity of the computational model may be further exacerbated by the multiple ways to obtain a piece of information. So, the model would have to capture information equivalency relations over the rich variety of sensor data, which in turn would require semantic information and services to process and reason about the sensor data in ways that move beyond protocol agreement and data-format conversion. To quantify the semantic information contained in sensor data and to capture the relations between various semantic entities such as objects and events in the physical world, the model would need to define an ontology for a variety of commonly encountered sensor data and provide a set of semantic transformation services to incrementally extract new semantic information from lower level data. (This topic is of particular interest to the author and the reader is referred to [5], [6], [7], and [8].)

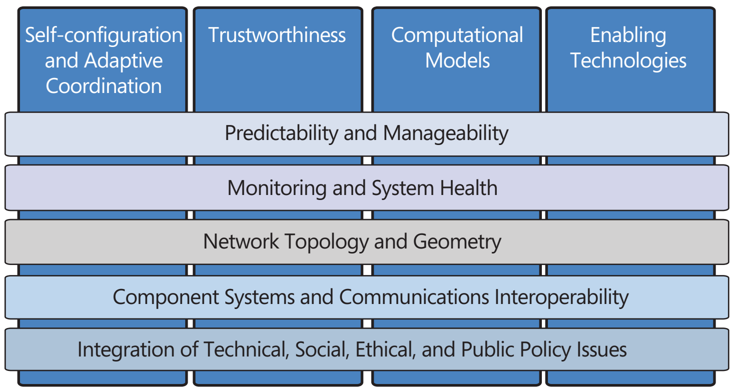
Enabling Technologies

The evolution of more and more sophisticated DES and complex embedded systems solutions is fueled by the revolutionary advances in information technology during the past several decades. Silicon scaling is the main driving force, with exponentially increasing processor performance enabling a world in which sophisticated chips can be manufactured and embedded easily and cheaply. Continued improvements in the price and performance of chip technology are expected for at least the next decade. Still, new developments are needed in specific aspects of communications, geo-location, software, and operating systems.

As wireless (and wired) technologies continue to become less expensive and more sophisticated, the vision of pervasive, connecting, embedded processors becomes increasingly feasible. However, most of the progress to date in wireless technology has focused on medium- to long-range communications (as in cellular phones and pagers) and is not sufficient for the widespread deployment of DES. Work is needed to understand how to create network architectures and designs for low-power, short-range wireless systems. Related to wireless are the issues surrounding geo-location technology. Unlike conventional computer networks, which are more dependent on the relative positioning of elements in a network topology, DES are often inextricably tied to the physical world (a primary purpose often being to measure and control physical-world attributes or objects), so location in physical space is more important. DES will therefore require ready access to absolute or relative geographic information, especially for sensor networks responsible for physical sensing and actuation.

Attention must also be given to the responsible and effective integration of sensor devices into DES, which will have impacts on the design of network-aware embedded operating systems, application development, and software tooling that has the characteristics required to tailor solutions to the physical constraints, afford in-place deployment, be upgradable, have high availability, and the ability to migrate to new hardware. Since DES will be embedded in long-

Figure 1: Technical Imperatives for Distributed Embedded Systems Development and Research


Self-configuration and Adaptive Coordination
Trustworthiness
Computational Models
Enabling Technologies
Predictability and Manageability
Monitoring and System Health
Network Topology and Geometry
Component Systems and Communications Interoperability
Integration of Technical, Social, Ethical, and Public Policy Issues

lived structures, they will have to evolve, depending on changing external conditions and advances in technology as time passes. Thus, operating systems and applications that can cope with this type of evolution will be critical. Furthermore, real-time operation and performance-critical constraints on multiple embedded applications running concurrently on the same device will create a demand for entirely new methods of software development and systems architectures.

Figure 1 is a chart of the technical imperatives for DES development and research discussed in the previous paragraphs. Additional issues present in this field are given in summary form and serve to highlight the significant differences from traditional distributed systems development.

Further Issues and Needs:

1. *Predictability and manageability* – Covers methodologies and mechanisms for designing predictable, safe, reliable, manageable distributed embedded systems.
2. *Monitoring and system health* – Refers to a complete conceptual framework to help achieve robust operation through self-monitoring, continuous self-testing, and reporting of system health in the face of extreme constraints on nodes and elements of the system.
3. *Network topology and geometry* – Represent modeling techniques to support and incorporate network geometry (as opposed to just network topology) into DES.
4. *Component systems and communications interoperability* – Is about the techniques and design methods for constructing long-lived, heterogeneous systems that evolve over time and space while remaining interoperable.
5. *Integration of technical, social, ethical, and public policy issues* – Is predominantly about the fundamental research required to address nontechnical issues of Embedded Systems, especially those having to do with the ethical and public policy issues surrounding privacy, security, reliability, usability, and safety.

Development

To understand the nature of embedded systems development complexity you should also understand the typical embedded device value chain. This consists of links between various industry players comprising of silicon vendors (SVs), embedded OS vendors, independent hardware vendors (IHVs), device makers (original equipment developer (OED) and original equipment or design manufacturer (OEM/ODM)), distributors and end customers (businesses or consumers). In Figure 2, you see the embedded device value chain consists of 3-7 links between SVs, OEDs, OEMs, and end customers. And depending on the device maker's supply chain and market, it may also consist of third-party software providers, IHVs, ODMs; and various distribution intermediaries.

When developing traditional business applications and systems, "devices" are standardized and readily available off-the-shelf computers, making a significant part of their value chain quite irrelevant to the development process. However, in embedded systems design and development, a similar level of maturity or commoditization, if you like, of "devices" has not yet been achieved in the industry. As a result, development tool chains, processes, and methodologies are often proprietary and established for a specific goal. The vast majority of the industry in embedded development is using open source software and custom development tools provided by the hardware vendors (SV, IHV) and open source communities. Time scales for the development of hardware and software at all levels in the device value chain can therefore be long as it requires proprietary or scarce skills and knowledge to put solutions together. In some cases, where DES and real-time scenarios are involved, especially in safety-critical situations, these time scales can easily span a decade. There are interesting offerings from the large operating systems and software tools vendors such as Microsoft that hold the promise of providing a more productive and "traditional" development experience for embedded systems professionals (including academics and hobbyists) in many key solution scenarios. The hope is that eventually tool chains along the entire device value chain will become mostly standardized and interoperable enabling the embedded systems industry to scale development as seen in traditional software development.

Figure 2: The Embedded Device Value Chain

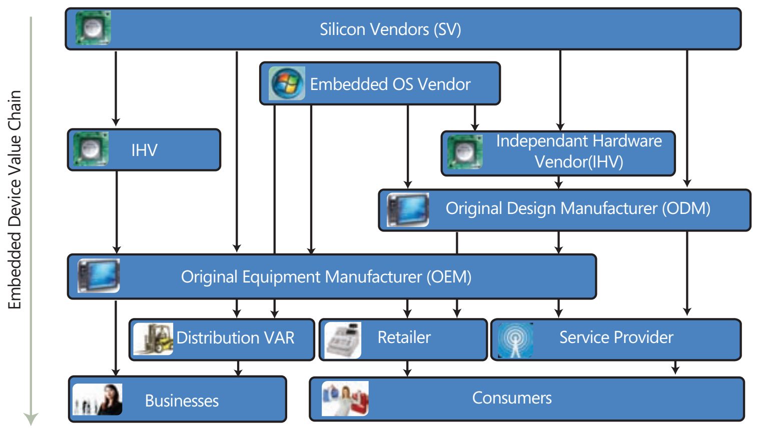

Embedded Device Value Chain
Silicon Vendors (SV)
Embedded OS Vendor
IHV
Independant Hardware Vendor(IHV)
Original Design Manufacturer (ODM)
Original Equipment Manufacturer (OEM)
Distribution VAR
Retailer
Service Provider
Businesses
Consumers

Figure 3: Embedded Systems Development Lifecycle and Microsoft Technologies

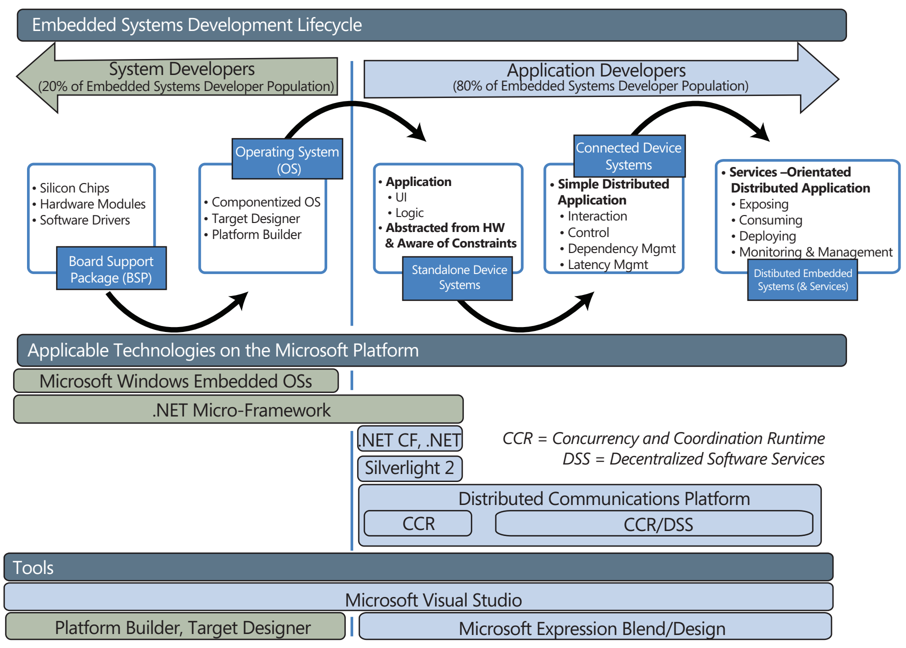

Embedded Systems Development Lifecycle
System Developers
(20% of Embedded Systems Developer Population)
Application Developers
(80% of Embedded Systems Developer Population)
• Silicon Chips
• Hardware Modules
• Software Drivers
Board Support Package (BSP)
Operating System (OS)
• Componentized OS
• Target Designer
• Platform Builder
• Application
• UI
• Logic
• Abstracted from HW & Aware of Constraints
Standalone Device Systems
Connected Device Systems
• Simple Distributed Application
• Interaction
• Control
• Dependency Mgmt
• Latency Mgmt
• Services –Orientated Distributed Application
• Exposing
• Consuming
• Deploying
• Monitoring & Management
Distibuted Embedded Systems (& Services)
Applicable Technologies on the Microsoft Platform
Microsoft Windows Embedded OSs
.NET Micro-Framework
.NET CF, .NET
Silverlight 2
CCR = Concurrency and Coordination Runtime
DSS = Decentralized Software Services
Distributed Communications Platform
CCR
CCR/DSS
Tools
Microsoft Visual Studio
Platform Builder, Target Designer
Microsoft Expression Blend/Design

Figure 3 shows some aspects of the embedded systems development life cycle—which is strongly tied to the device value chain—and the Microsoft platform technologies that may be applied through the lifecycle.

Because embedded devices contain both hardware and software components understanding the interactions between them are essential. Key skills in this area span electrical engineering and computer science; they also address the interrelationships among processor architectures, performance optimization, operating systems, virtual memory, concurrency, task scheduling, and synchronization. Developers should have a breadth of exposure to appreciate the utility of other fields in embedded systems, such as digital signal processing and feedback control. Finally, as embedded systems are not general operating systems, developers should understand the tight coupling between embedded applications and the hardware platforms that host them.

Conclusion

This paper introduces you to the rich field of distributed embedded systems. It is geared towards architects and developers of traditional distributed systems and therefore takes a broad perspective on the task of describing DES. Several references at the end may assist readers in exploring this subject in greater depth. Distributed systems design approaches and their characteristics will influence DES design and implementation, but there are significant differences that have been described in this paper. The way we conceive and program these systems is bound to become easier. It has to, because smart, connected, service-oriented embedded systems will permeate our factories, hospitals, transportation systems, entertainment, and personal spaces, and thus become essential to our daily lives. In many ways, DES is the computing model for the future, and the mainstream of the IT industry will need to learn how to design, build, and operate these systems well.

References

1 Dan Nessett (1999), *Massively Distributed Systems: Design Issues and Challenges*, USENIX Workshop on Embedded System, 1999.

2 FAST Report, *Study of Worldwide Trends and R&D Programmes in Embedded Systems*, 2005.

3 Siegemund F., Haroon M., Ansari J., Mahonen P., *Senslets – Applets for the Sensor Internet*, Wireless Communications and Networking Conference, 2008.

4 National Research Council, Committee on Networked Systems of Embedded Computers, Embedded, *Everywhere: A Research Agenda for Networked Systems of Embedded Computers*, National Academy Press, Washington, D.C., 2001.

5 Jie Liu and Feng Zhao, Towards Semantic Services for Sensor-Rich Information Systems, 2nd International Conference on Broadband Networks, 2005.

6 Jie Liu, Eker, J., Janneck, J.W., Xiaojun Liu, Lee, E.A., Actor-Oriented Control System Design: A Responsible Framework Perspective, IEEE Transactions on Control Systems Technology, 12(2), March 2004.

7 Schmidt D. C., Gokhale A., Schantz R. E., and Loyall, J. P., *Middleware R&D Challenges for Distributed Real-Time and Embedded Systems*, ACM SIGBED Review, 1 (1), 2004.

8 Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn l. Talcott, A *Foundation for Actor Computation*, J. Functional Programming, 7 (1), 1997.

About the Author

Arvindra Sehmi is a director of Developer and Platform Evangelism at Microsoft. His team works with customers on business and application architecture methodology and best-practices adoption. Currently, he is working in the area of distributed embedded systems and its role in Microsoft's Software + Services strategy and Cloud Computing platform. Previously as lead Architect Evangelist in EMEA, he was recognized by Microsoft for outstanding achievement in this profession. Arvindra has specialized in the financial services industry and created the Microsoft Architecture Journal for which he remains its editor emeritus. He holds a Ph.D. in bio-medical engineering and masters in business. You can find his blog at http://blogs.msdn.com/asehmi and contact him at arvindra.sehmi@microsoft.com.

Using Events in Highly Distributed Architectures

by David Chou

Introduction

Many service-oriented architecture (SOA) efforts today are focusing on implementing synchronous request-response interaction patterns (sometimes using asynchronous message delivery) to connect remote processes in distributed systems. While this approach works for highly centralized environments, and can create loose coupling for distributed software components at a technical level, it tends to create tight coupling and added dependencies for business processes at a functional level. Furthermore, in the migration towards real-time enterprises, which are also constantly connected and always available on the Web, organizations are encountering more diverse business scenarios and discovering needs for alternative design patterns in addition to synchronous request-driven SOA.

Event-driven architecture (EDA, or event-driven SOA) is often described as the next iteration of SOA. It is based on an asynchronous message-driven communication model to propagate information throughout an enterprise. It supports a more "natural" alignment with an organization's operational model by describing business activities as series of events, and does not bind functionally disparate systems and teams into the same centralized management model. Consequently, event-driven architecture can be a more suitable form of SOA for organizations that have a more distributed environment and where a higher degree of local autonomy is needed to achieve organizational agility.

This article discusses how EDA, together with SOA, can be leveraged to achieve a real-time connected system in highly distributed environments, and be more effective at understanding organizational state. We will then discuss a pragmatic approach in designing and implementing EDA, while considering the trade-offs involved in many aspects of the architecture design.

Event-Driven Architecture

So what do we mean by events? Well, events are nothing new. We have seen various forms of events used in different areas such as event-driven programming in modern graphical user interface (GUI) frameworks, SNMP and system notifications in infrastructure and operations, User Datagram Protocol (UDP) in networking, message-oriented middleware (MOM), pub/sub message queues, incremental database synchronization mechanisms, RFID readings, e-mail messages, Short Message Service (SMS) and instant messaging, and so on.

In the context of SOA however, an event is essentially a significant or meaningful change in state. And an event in this perspective takes a higher level semantic form where it is more coarse-grained, and abstracted into business concepts and activities. For example, a new customer registration has occurred on the external Website, an order has completed the checkout process, a loan application was approved in underwriting, a market trade transaction was completed, a fulfillment request was submitted to a supplier, and so on. Fine-grained technical events, such as infrastructure faults, application exceptions, system capacity changes, and change deployments are still important but are considered more localized in terms of scope, and not as relevant from an enterprise-level business alignment perspective.

As event sources publish these notifications, event receivers can choose to listen to or filter out specific events, and make proactive decisions in near real-time about how to react to the notifications. For example, update customer records in internal systems as a result of a new customer registration on the Website, update the billing and order fulfillment systems as a result of an order checkout, update customer account as a result of a back office process completion, transforming and generating additional downstream events, and so on.

Event-driven architecture is an architectural style that builds on the fundamental aspects of event notifications to facilitate immediate information dissemination and reactive business process execution.

In an event-driven architecture, information can be propagated in near-real-time throughout a highly distributed environment, and enable the organization to proactively respond to business activities. Event-driven architecture promotes a low latency and highly reactive enterprise, improving on traditional data integration techniques such as batch-oriented data replication and posterior business intelligence reporting. Modeling business processes into discrete state transitions (compared to sequential process workflows) offers higher flexibility in responding to changing conditions, and an appropriate approach to manage the asynchronous parts of an enterprise.

Figure 1: Request-driven architecture (logical view)

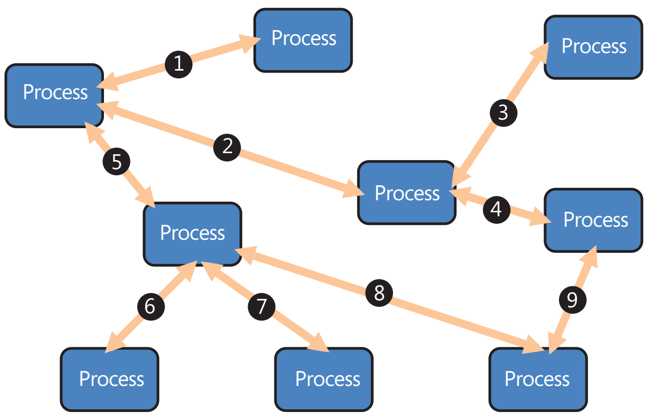
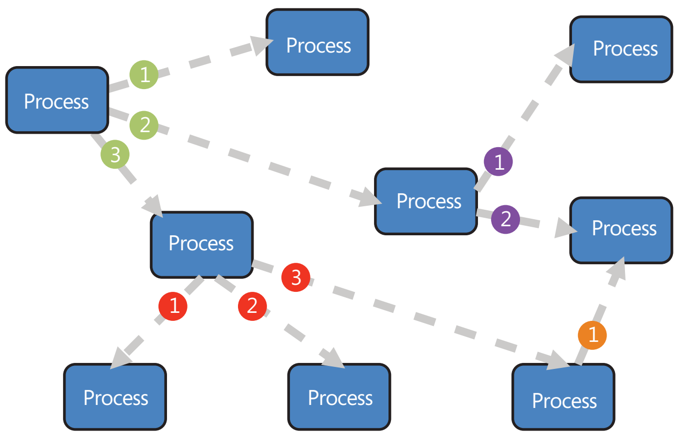


Figure 2: Event-driven architecture (logical view)



Fundamental Principles

The way systems and business processes are connected in an event-driven architecture represents another paradigm shift from request-driven SOA. Here we will discuss some of the fundamental aspects.

Asynchronous "Push"- Based Messaging Pattern

Instead of the traditional "pull"- based synchronous request/response or client/server RPC-style interaction models, event-driven architecture builds on the pub/sub model to "push" notifications out to interested listeners, in a "fire-and-forget" (does not block and wait for a synchronous response), uni-directional, asynchronous pattern.

Autonomous Messages

Events are transmitted and communicated in the form of autonomous messages — each message contains just enough information to represent a unit of work, and to provide decision support for notification receivers. Each event message also should not require any additional context, or any dependencies on the in-memory session state of the applications; it is intended to communicate the business state transitions of each application, domain, or workgroup in an enterprise.

For example, an externally-facing Web application publishes an event that a new customer registration was completed with these additional attributes: timestamp, ID=123, type=business, location=California, email=jane@contoso.com; information that is relevant to the rest of the enterprise, and supports any process-driven synchronous lookups for additional details to published data services from distributed systems.

Higher decoupling of distributed systems

This asynchronous message-based interaction model further encapsulates internal details of distributed systems in an environment. This form of communication does not need to be as concerned with some aspects of request-response models such as input parameters, service interface definitions such as WSDL in SOAP-based communication and hierarchical URI definitions in REST-based communication, and fine-grained security. The only requirement is a well-defined message semantic format, which can be implemented as plain old XML-based (POX) payload.

The event-driven pattern also logically decouples connected systems, compared to the technical loose-coupling achieved in the request-driven pattern. That is, functional processes in the sender system, where the business event took place, do not depend on the availability and completion of remote processes in downstream distributed systems. Whereas in a request-driven architecture, the sender system needs to know exactly which distributed services to invoke, exactly what the provider systems need in order to execute those services, and depends on the availability and completion of those remote functions in order to successfully complete its own processing.

Furthermore, the reduced logical dependencies on distributed components also have similar implications on the physical side. In synchronous request-driven architectures, connected and dependent systems are often required to meet the service-level requirements (i.e., availability and throughput) and scalability/elasticity of the system that has the highest transaction volume. But in asynchronous event-driven architectures, the transaction load of one system does not need to influence or depend on the service levels of downstream systems, and allows application teams to be more autonomous in designing their respective physical architectures and capacity planning.

This also means that changes to each connected system can be deployed more independently (and thus more frequently) with minimal impact to other systems, due to the overall reduced dependencies.

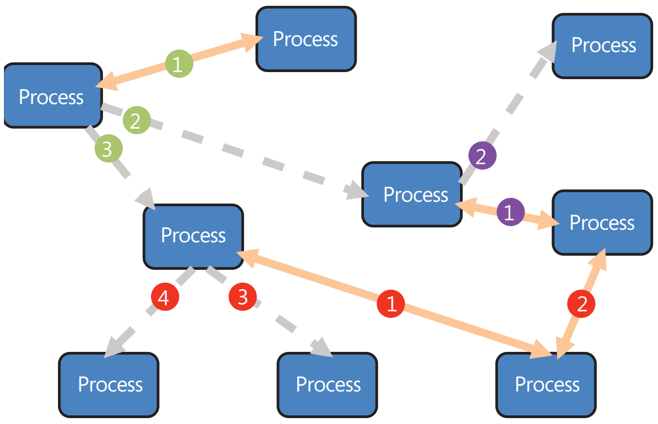
In the case of an externally facing Web application that has just completed a new customer registration process, that information needs to be communicated to an internal customer management system. The Web application's only concern is to publish the event notification according to previously defined format and semantics; it does not need to remotely execute a service in the customer management system to complete its own processing, or bind the customer management system into a distributed transaction to make sure that the associated customer data is properly updated. The Web application leaves the downstream customer management system to react to the event notification and perform whatever it needs to do with that information.

Receiver - Driven Flow Control

The event-driven pattern shifts much of the responsibility of controlling flow away from the event source (or sender system), and distributes/delegates it to event receivers. Event source systems do not need to explicitly define the scope of distributed transactions and manage them, or connected systems are less dependent on centralized infrastructure components such as transaction process systems (TPS), business process management (BPM), and enterprise service bus (ESB) products. The connected systems participating in an event-driven architecture have more autonomy in deciding whether to further propagate the events, or not. The knowledge used to support these decisions is distributed into discrete steps or stages throughout the architecture, and encapsulated where the ownerships reside, instead of burdening the transaction-initiating sender systems with that knowledge.

For example, in the new customer registration event scenario, the customer management system that receives the event notification owns the decision on executing a series of its own workflows, populating its own database of this information, and publishing a potentially different event that the marketing department may be interested in. All these downstream activities rely on decisions owned and made at each node, in a more independent and collective model.

Figure 3: Hybrid request and event-driven architecture) logical view)



Near Real-Time Propagation

Event-driven architecture is only relevant when events are published, consumed, and propagated in near real-time. If this approach is applied in passive event-processing scenarios such as batch-oriented fixed schedule, then its effectiveness is not much different from the traditional data integration techniques used today.

This is primarily due to a simple fact – events "in-motion" are much more valuable than events "at rest". Gaining accurate and near real-time insight has been a complex and difficult challenge for enterprises; even with the advent of SOA, it is still relatively complicated given the existing focus on integrating distributed processes in synchronous interaction models. Event-driven architecture provides a comparatively simpler approach to achieve near real-time information consistency, as we just need to apply a different form of system interaction model using existing technologies and infrastructure investments.

Potential Benefits

By now, some potential benefits of using an event-driven architecture pattern may have become more apparent.

Effective data integration — In synchronous request-driven architectures, focus is typically placed on reusing remote functions and implementing process-oriented integration. Consequently, data integration is not well supported in a process-oriented model, and is often an overlooked aspect in many SOA implementations. Event-driven architecture is inherently based on data integration, where the events disseminate incremental changes in business state throughout the enterprise; thus it presents an effective alternative of achieving data consistency, as opposed to trying to facilitate data integration in the process layer.

Reduced information latency — As distributed systems participate in an event-driven architecture, event occurrences are enabled to ripple throughout the connected enterprise in near real-time. This is analogous to an enterprise nervous system where the immediate propagation of signals and notifications form the basis of near real-time business insight and analytics.

Enablement of accurate response — As business-critical data propagates throughout the enterprise in a timely manner, operational systems can have the most accurate, current view of the state of business. This enables workgroups and distributed systems to provide prompt and accurate business-level responses to changing conditions. This also moves us closer to being able to perform predictive analysis in an ad hoc manner.

Improved scalability — Asynchronous systems tend to be more scalable than synchronous systems. Individual processes block less and have reduced dependencies on remote/distributed processes. Plus, intermediaries can be more stateless, reducing overall complexity. Similar points can be made for reliability, manageability, and so forth.

Improved flexibility — Discrete event-driven processes tend to be more flexible in terms of the ability to react to conditions not explicitly defined in statically structured sequential business processes, as state changes can be driven in an ad hoc manner. And as mentioned earlier, the higher level of decoupling between connected systems means changes can be deployed more independently and thus more frequently.

Improved business agility — Finally, the ultimate goal of SOA implementations. Event-driven architecture promises enhanced business agility as it provides a closer alignment with operational models, especially in complex enterprise environments consisting of diverse functional domains and requiring high degrees of local autonomy. Plus, business concepts and activities are modeled in simpler terms with responsibilities appropriately delegated to corresponding owners. This creates an environment where decisions and business processes in one area are less dependent on the overall enterprise environment, compared to synchronous request-driven architecture's centralized view of logically coupled, sequentially structured business processes, where one component is often dependent on a number of distributed components for its processing. The reduced dependencies allow technical and functional changes to be planned and released more independently and frequently, achieving a higher level of IT agility and hence, business agility.

Aspects of Implementation

At a high level, event-driven architecture models business operations and processes using a discrete state machine pattern, whereas request-driven architecture tend to use more structurally static sequential flows. This results in some fundamental differences between the two architectural styles, and how to design their implementation approaches. However, the two architectural styles are not intended to be mutually exclusive; they are very complementary to each other as they address different aspects of operational models in an organization. Furthermore, it is the combination of request-driven and event-driven architectures that makes the advanced enterprise capabilities achievable. Here we will discuss some of the major consideration areas, from the perspective of building on top of existing service-oriented architecture principles.

Event Normalization and Taxonomy

First and foremost, event-driven architecture needs a consistent view of events, with clearly defined and standardized taxonomy so that the events are meaningful and can be easily understood—just as when creating enterprise-level standards of data and business process models, an enterprise-level view of events needs to be defined and maintained.

Process Design and Modeling

Modeling business processes into discrete event flows is similar to defining human collaboration workflows using state machines. A key is to distinguish between appropriate uses of synchronized request-driven processes or asynchronous event-driven processes.

For example, one organization used the synchronous request-driven pattern to implement a Web-based user registration process, in terms of populating data between multiple systems as a result of the registration request. The implementation leveraged an enterprise service bus solution, and included three systems in a synchronized distributed transaction. The resulting registration process waited until it received "commit successful" responses from the two downstream systems, and often faulted due to remote exceptions. It was a nonscalable and error-prone solution that often negatively impacted end users for a seemingly simple account registration process. On the other hand, as discussed previously, this type of scenario would obtain much better scalability and reliability if an asynchronous event-driven pattern was implemented.

However, if in this case the user registration process requires an identity proofing functionality implemented in a different system, in order to verify the user and approve the registration request, then a synchronous request-response model should be used.

As a rule of thumb, the synchronous request-driven pattern is appropriate when the event source (or client) depends on the receiver (or server) to perform and complete a function as a component of its own process execution; typically focused on reusing business logic. On the other hand, the asynchronous event-driven pattern is appropriate when connected systems are largely autonomous, not having any physical or logical dependencies; typically focused on data integration.

Communication

Event-driven architecture can leverage the same communication infrastructure used for service-oriented architecture implementations. And just the same as in basic SOA implementations, direct and point-to-point Web services communication (both SOAP and REST-based) can also be used to facilitate event-driven communication. And standards such as WS-Eventing and WS-Notification, even Real Simple Syndication (RSS) or Atom feeds can be used to support delivery of event notifications.

However, intermediaries play a key role in implementing an event-driven architecture, as they can provide the proper encapsulation, abstraction, and decoupling of event sources and receivers, and the much-needed reliable delivery, dynamic and durable subscriptions, centralized management, pub/sub messaging infrastructure, and so on. Without intermediaries to manage the flow of events, the communication graphs in a point-to-point eventing model could become magnitudes more difficult to maintain than a point-to-point service-oriented architecture.

Event Harvesting

The use of intermediaries, such as the modern class of enterprise service bus solutions, message-oriented middleware infrastructure, and intelligent network appliances, further enhance the capability to capture in-flight events and provide business-level visibility and insight. This is the basis of the current class of business activity monitoring (BAM) and business event management (BEM) solutions.

Security

Security in event-driven architecture can build on the same technologies and infrastructures used in service-oriented architecture implementations. However, because of the highly decoupled nature of event-driven patterns, event-based security can be simpler as connected systems can leverage a point-to-point, trust-based security model and context, compared to the end-to-end security contexts used in multistep request-driven distributed transactions.

Scope of Data

So how much data should be communicated via events? How is this different from traditional data replication? Data integration is intrinsic to event-driven architecture, but the intention is to inform other system "occurrences" of significant changes, instead of sending important data as events over the network. This means that only the meaningful data that uniquely identifies an occurrence of a change should be part of the event notification payload, or in other words, just an adequate amount of data to help event receivers decide how to react to the event. In the example discussed earlier with the customer registration event, the event message does not need to contain all user data elements captured/used in the customer registration process, but the key referential or identifying elements should be included. This helps to maintain clear and concise meaning in events, appropriate data ownerships, and overall architectural scalability. If downstream systems require more data elements or functional interaction to complete their processes, alternative means, such as synchronous Web services communication, can be used to facilitate that part of the communication.

State of the Art

Here we discuss several emerging trends that influence advanced applications of event-driven processing.

Complex Event Processing

So far we have only explored using individual events as a means of information dissemination and achieving enterprise-wide business state consistency across highly distributed architectures. Multiple events can be interpreted, correlated, and aggregated based on temporal, historical, and/or computational algorithms to discern meaningful patterns. Basically, in addition to the information individual events carry, combinations of multiple events (or the lack of) can provide significant meaning.

Similar capabilities can be found in today's SOA implementations, but are typically implemented locally to a system examining a request against historical and environmental data at rest, where visibility is limited and interpretation rules are encapsulated. But the use of service intermediaries provides centralized event management, higher visibility, and more relevant interpretation of event streams. Also, as complex event processing often makes use of rules engines to drive the pattern recognition behaviors, changes to how events are processed can be deployed quickly, allowing business decisions to be implemented quickly.

This information can also be derived into composite or synthetic events to prompt downstream actions. For example, in practice today, complex event processing is often used in fraud detection to identify pattern consistency (e.g., banking transactions conducted in the same geography), pattern anomaly (e.g., a transaction requested from a different geography when other requests are consistently within the same geography), and even lack of events. Once the fraud detection rule identifies a potential attack, the monitoring system can send that information as another event to alert other systems in the enterprise, where specific actions can be taken in response to this event.

Events in the Cloud

As we move towards cloud computing, event processing can also be leveraged to capture and correlate event streams running in the open cloud. However, unlike event-driven architectures within an enterprise, where the focus is organizing business state changes in distributed systems into a consistent view, integrating event streams from the public cloud follows a much less structured form and approach, as the data and context in publicly available events are highly heterogeneous. Information latencies are also expected to be higher in the open cloud. From an implementation perspective, tapping into public event streams could be as simple as subscribing to RSS or Atom feeds directly from other Websites and organizations (including feed aggregators), direct point-to-point Web services integration (either via SOAP-based SOA or REST-based WOA approaches), or facilitated through the emerging class of cloud-based "Internet service bus" intermediaries (such as Microsoft's Cloud Services platform) where message delivery, service levels, identity and access control, transactional visibility, and contextualized taxonomies can be more formally managed between multiple enterprises. Consequently, the interpreted information from cloud-based event streams can be correlated against event streams internal to an enterprise. The result is a highly reactive enterprise that is cognitive of market and environmental changes as they occur, in which business decisions (whether manual or automated) can be made with significantly higher relevancy.

Context - Aware Computing

The next perceived step in business application architecture is towards context-aware computing. Context from this perspective refers to information that is not explicitly bound to the data elements of a particular process or transaction, but is relevant via implicit relationships. For example, geo-location, time (of the day, week, month, seasons, holidays, special occasions), real-life events such as presidential elections, Olympic games (e.g., subsequent potential impact on retailers and their supply chains), stock market conditions, interest rates, raw material over supply or shortage, new media blockbuster hit, new invention, and even weather, can all be information (or event occurrences) that influence human behaviors and decisions, which also convert into interactions with online systems. This contextual information is useful as it provides additional decision support factors, but most importantly, aids towards understanding user intent, such that different responses can be generated for similar requests, influenced by different contextual factors. As a result, context-aware systems are much more connected with the environment and thus can be more relevant, compared to today's systems that are mostly silo'ed within the limited views of their own databases.

Event-driven architecture can play a key role in facilitating context-aware computing, as it is the most effective and scalable form of data integration for highly distributed architectures. As discussed previously in this article, higher level decoupling of distributed systems and reduced dependencies in event-driven architectures can significantly simplify efforts in harvesting and interpreting business events within an enterprise, plus tapping into massive amounts of cloud-based events. As a result, event-driven architecture can help systems keeping an accurate pulse of its relevant environments, and becoming more context-aware.

Conclusion

Event-driven architecture brings about the convergence of disparate forms of internal and external events, such as system and IT management notifications, business activities and transactions, public market, technical, and social events. The organization of these events into a unified and consistent view promises enhanced alignment between business and IT, and moves towards a connected, dynamic enterprise that has accurate and consistent business insight, and can proactively respond to changing conditions in near real-time.

However, asynchronous event-driven architecture is not intended to be a replacement of existing forms of synchronous request-driven service-oriented architectures. In fact, they are very complementary to each other when appropriately modeling the different aspects of an organization's operational model. From a technical perspective, event-driven architecture can be implemented on top of an existing service-oriented infrastructure, without adding more dependencies and complexities. An effective mix of both architectural patterns will create a highly consistent and responsive enterprise, and establish a solid foundation for moving towards always-on and always-connected environments.

Resources

"Complex Event Processing in Distributed Systems," David C. Luckham and Brian Frasca, Stanford University Technical Report CSL-TR-98-754, http://pavg.stanford.edu/cep/fabline.ps.

"Event-Based Execution Architectures for Dynamic Software Systems," James Vera, Louis Perrochon, David C. Luckham, Proceedings of the First Working IFIP Conf. on Software Architecture, http://pavg.stanford.edu/cep/99wicsa1.ps.gz.

"Complex Event Processing: An Essential Technology for Instant Insight into the Operation of Enterprise Information Systems," David C. Luckham, lecture at the Stanford University Computer Systems Laboratory EE380 Colloquium series, http://pavg.stanford.edu/cep/ee380abstract.html.

"Service-Oriented Architecture: Developing the Enterprise Roadmap," Anne Thomas Manes, Burton Group, http://www.burtongroup.com/Client/Research/Document.aspx?cid=136.

"The Unification of SOA and EDA and More," Randy Heffner, Forrester, http://www.forrester.com/Research/Document/Excerpt/0,7211,35720,00.html.

"Clarifying the Terms 'Event-Driven' and 'Service-Oriented' Architecture," Roy W. Shulte, Gartner, http://www.gartner.com/DisplayDocument?doc_cd=126127&ref=g_fromdoc.

"Event-Driven Architecture Overview," Brenda M. Michelson, Patricia Seybold Group, http://www.psgroup.com/detail.aspx?id=681.

About the Author

David Chou is an architect in the Developer & Platform Evangelism group at Microsoft, where he collaborates with local organizations in architecture, design, and proof-of-concept projects. David enjoys helping customers create value by using objective and pragmatic approaches to define IT strategies and solution architectures. David maintains a blog at http://blogs.msdn.com/dachou, and can be reached at david.chou@microsoft.com.

Caching in the Distributed Environment

by Abhijit Gadkari

Summary

The impact of cache is well understood in the system design domain. While the concept of cache is extensively utilized in the von Neumann architecture, the same is not true for the distributed computing architecture. For example, consider a three-tiered web-based business application running on a commercial RDBMS. Every time a new web page loads, many database calls are made to fill the drop down lists on the page. Performance of the application is greatly impacted by the unnecessary database calls and the network traffic between the web server and the database server.

In production, many applications buckle down because they treat the database as their cache. Web server-based application-level cache can be effectively used to mitigate this problem. An effective caching mechanism is the foundation of any distributed computing architecture. The focus of this article is to understand the importance of caching in designing effective and efficient distributed architecture. I will discuss the principle of locality of cache, basic caching patterns like temporal and spatial cache, and primed and demand cache, followed by an explanation of the cache replacement algorithms.

ORM technologies are becoming part of the mainstream application design, adding a level of abstraction. Implementing ORM-level cache will improve the performance of a distributed system. I will explain different ORM caching levels such as transactional cache, shared cache, and the details of intercache interaction. I'll also explore the impact of ORM caching on application design.

Distributed Systems

A distributed system is a heterogeneous system. Diverse system components are connected to each other via a common network. Applications using TCP/IP-based Internet are examples of open distributed systems. Figure 1 shows a typical distributed architecture.

In the distributed environment, different activities occur in concurrent fashion. Usually, common resources like the underlying network, web/application servers, database servers, and cache servers are shared by many clients. Distributing the computing load is the hallmark of distributed systems. Resource sharing and allocation is a major challenge in designing distributed architecture. For example, consider a web-based database-driven business application. The web server and the database server are hammered with client requests. Caching, load-balancing, clustering, pooling, and time-sharing strategies improve the system performance and availability. I'll focus on caching in the distributed environment.

Any frequently consumed resource can be cached to augment the application performance. For example, caching a database connection, an external configuration file, workflow data, user preferences, or frequently accessed web pages improve the application performance and availability. Many distributed computing platforms offer out-of-the-box caching infrastructure. Java Caching System (JCS) is a distributed composite caching system. In .NET, the System.Web.Caching API provides the necessary caching framework. Microsoft project code named Velocity is a distributed caching platform [1].

The performance of a caching system depends on the underlying caching data structure, cache eviction strategy, and cache utilization policy. Typically, a hash table with unique hash keys is used to store the cached data; JCS is a collection of hash tables [2]. The .NET cache implementation is based on the Dictionary data structure. The cache eviction policy is implemented in terms of a replacement algorithm. Utilizing different strategies such as temporal, spatial, primed, and demand caching can create an effective caching solution.

Cache and the Principle of Locality

The word "cache" comes from the French meaning "to hide" [3]. Wikipedia defines cache as "a temporary storage area where frequently accessed data can be stored for rapid access" [4]. Cached data is stored in the memory. Defining frequently accessed data is a matter of judgment and engineering. We have to answer two fundamental questions in order to define a solid caching strategy. What resource should be stored in

Figure 1: Distributed architecture

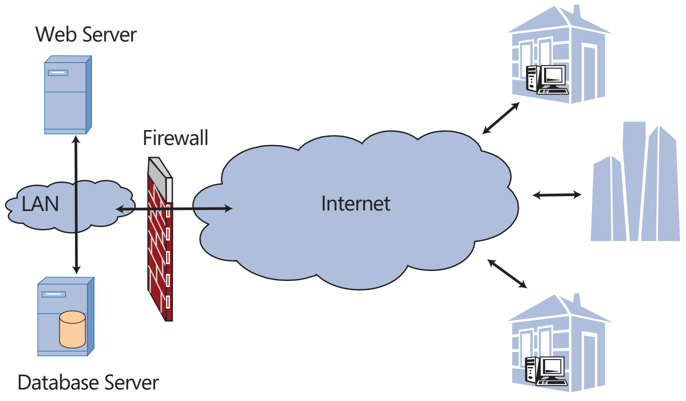
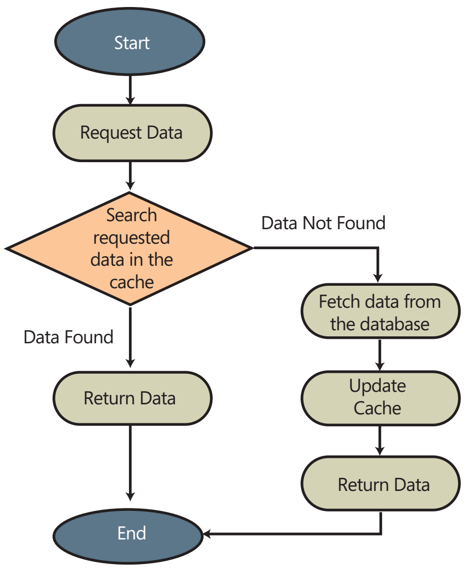


Figure 2: Temporal locality flowchart



the cache? How long should the resource be stored in the cache? The locality principle, which came out of work done on the Atlas System's virtual memory in 1959 [5], provides good guidance on this front, defining temporal and spatial locality. Temporal locality is based on repeatedly referenced resources. Spatial locality states that the data adjacent to recently referenced data will be requested in the near future.

Temporal Cache

Temporal locality is well suited for frequently accessed, relatively nonvolatile data; for example, a drop-down list on a web page. The data for the drop down list can be stored in the cache at the start of the application on the web server. For subsequent web page requests, the drop down list will be populated from the web server cache and not from the database. This will save unnecessary database calls and will improve application performance. Figure 2 illustrates a flow chart for this logic.

When a resource is added to the cache, resource dependencies can be added to the caching policy. Dependencies can be configured in terms of an external file or other objects in the cache. An expiration policy defines the time dependency for the cached resource. Many caching APIs provide a programmatic way to synchronize the cache with the underlying database. Figure 3 is sample C# code to populate the temporal cache.

Spatial Cache

Consider an example of tabular data display like a GridView or an on-screen report. Implementing efficient paging on such controls requires complex logic. The logic is based on the number of records displayed per page and the total number of matching records in the underlying database table. We can either perform in-memory paging or hit the database every time the user moves to a different page – both are extreme scenarios. A third solution is to exploit the principle of spatial locality to implement an efficient paging solution.
For example, consider a GridView displaying 10 records per page. For 93 records, we will have 10 pages. Rather than fetching all records in the memory, we can use the spatial cache to optimize this process. A sliding window algorithm can be used to implement the paging. Let's define the data window just wide enough to cover most of the user requests, say 30 records. On page one, we will fetch and cache the first 30 records. This cache entry can be user session specific or applicable across the application. As a user browses to the third page, the cache will be updated by replacing records in the range of 1-10 by 31-40.
In reality, most users won't browse beyond the first few pages. The cache will be discarded after five minutes of inactivity, eliminating the possibility of a memory leak. The logic is based on the spatial dependencies in the underlying dataset. This caching strategy works like a charm on a rarely changing static dataset. Figure 4 illustrates the spatial cache logic used in the GridView example.

The drawback of this logic is the possibility of a stale cache. A stale cache is a result of the application modifying the underlying dataset without refreshing the associated cache, producing inconsistent results. Many caching frameworks provide some sort of cache synchronization mechanism to mitigate this problem. In .NET, the SqlCacheDependency class in the System.Web.Caching API can be used to monitor a specific table [6]. SqlCacheDependency refreshes the associated cache when the underlying dataset is updated.

Cache Replacement Algorithms

A second important factor in determining an effective caching strategy is the lifetime of the cached resource. Usually, resources stored in the temporal cache are good for the life of an application. Resources stored in the spatial cache are time- or place-dependent. Time-dependent resources should be purged as per the cache

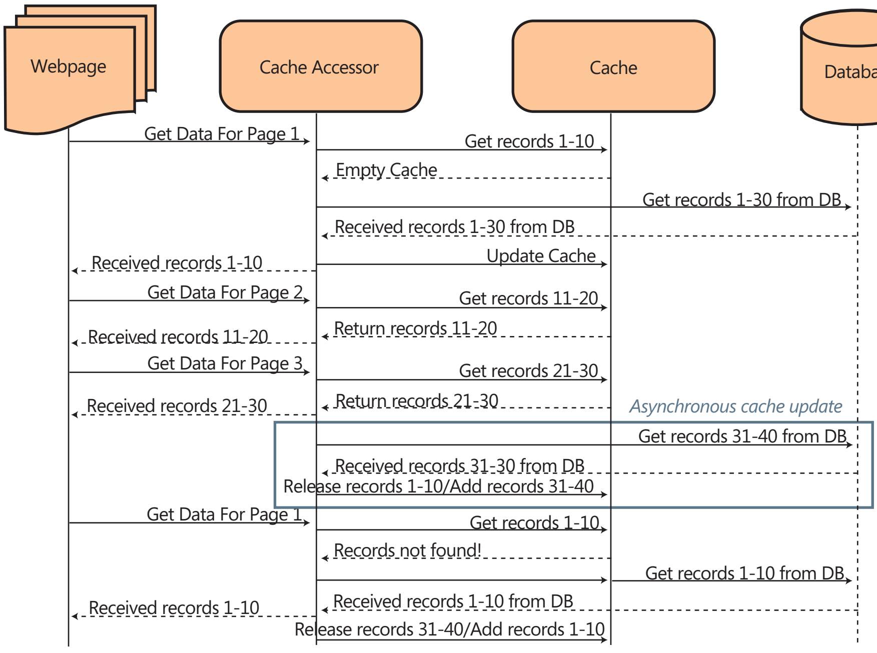
Figure 3: C# code example to populate temporal cache

```
protected void FillDepartmentList()
    {
      DataTable dtDepartment = (DataTable)Cache["departmentlist"];

      try
      {
        //if the cache is empty, get data from the database
        //and populate the cache
        if (dtDepartment == null)
        {
          dtDepartment = FillDropDownList.FillDepartmentListFromDB();
          Cache["departmentlist"] = dtDepartment;
        }

        //Make sure the cache is not empty
        if (dtDepartment.Rows.Count > 0)
        {
          ddlDepartmentList.DataSource = dtDepartment;
          ddlDepartmentList.DataBind();
        }
        else
        {
          ddlDepartmentList.Items.Add(new ListItem("No Department found!",
string.Empty));
        }
      }
      catch (Exception exce)
      {
        Response.Write(exce.Message);
      }
    }
```

Figure 4: Spatial cache sequence diagram



expiration policy. Place-specific resources can be discarded based on the state of the application. In order to store a new resource in the cache, an existing cached resource will be moved out of the cache to a secondary storage, such as the hard disk. This process is known as paging. Replacement algorithms such as least frequently used resource (LFU), least recently used resource (LRU), and most recently used resource (MRU) can be applied in implementing an effective cache-eviction strategy, which influences the cache predictability [7]. The goal in implementing any replacement algorithm is to minimize paging and maximize the cache hit rate. The cache hit rate is the possibility of finding the requested resource in the cache. In most cases, LRU implementation is a good enough solution. JCS and ASP. NET caching is based on the LRU algorithm. In more complex scenarios, a combination of LRU and LFU algorithms such as the adaptive replacement cache (ARC) can be implemented. The idea in ARC is to replace the least frequently and least recently used cached data. This is achieved by maintaining two additional scoring lists. These lists will store the information regarding the frequency and timestamp of the cached resource. ARC outperforms LRU by dynamically responding to the changing access pattern and continually balancing workload and frequency features [8]. Some applications implement a cost-based eviction policy. For example, in SQL Server 2005, zero-cost plans are removed from the cache and the cost of all other cached plans is reduced by half [9]. The cost in SQL Server is calculated based on the memory pressure.
A study of replacement algorithms suggests that a good algorithm should strike a balance between the simplicity of randomness and the complexity inherent in cumulative information [10]. Replacement algorithms play an important role in defining the cache-eviction policy, which directly affects the cache hit-rate and the application performance.

Primed vs. Demand Cache – Can We Predict the Future?

Data usage predictability also influences the caching strategy. The primed cache pattern is applicable when the cache or part of the cache can be predicted in advance [11]. This pattern is very effective in dealing with static resources. A web browser cache is an example of primed cache – cached web pages will load fast and save trips to the web server. The demand cache pattern is useful when cache cannot be predicted [12]. A cached copy of user credentials is an example of demand cache. The primed cache is populated at the beginning of the application, whereas the demand cache is populated during the execution of the application.

Primed Cache

The primed cache minimizes the overhead of requesting external resources. It is suitable for the read-only resources frequently shared by many concurrent users. Figure 5 illustrates the typical primed cache architecture.

The cache server cache is primed in advance, and the individual web/ application server cache is populated from the cache server. Each web/ application server can read, write, update, and delete the cache on the cache server. The cache server in turn is responsible for synchronizing the cache with the resource environment. Since the primed cache is populated in advance, it improves the application response time. For example, reports with static, fixed parameters can be populated and stored in the cache. This way, the reports are available almost instantly. In .NET, the ICachedReport interface can be used to store the prepopulated reports in

Figure 5: Primed cache example

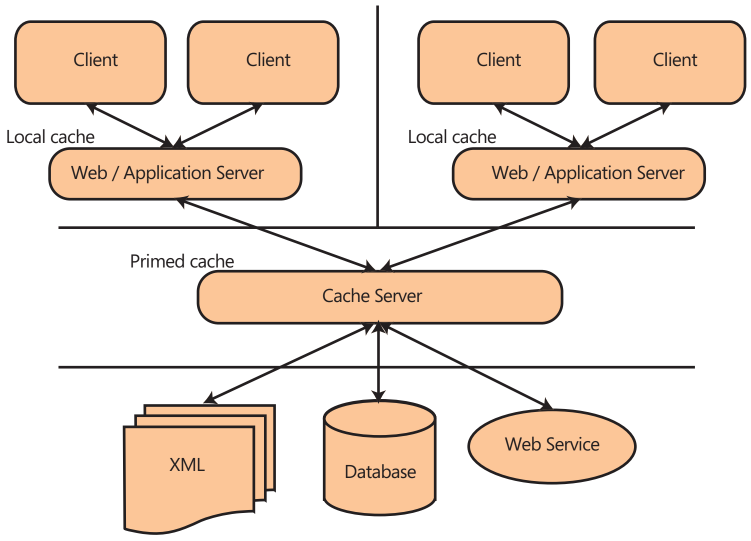
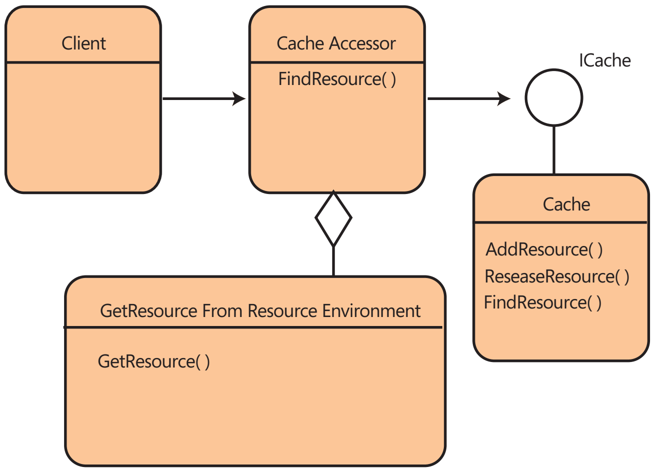


Figure 6: Demand cache



the cache. Updating the primed cache mostly results in updating existing cached resources. The cache is refreshed based on a routine schedule or a predictable event-based mechanism. The primed cache results in an almost constant size cache structure [11].

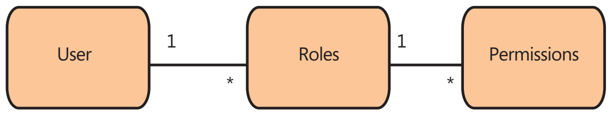
Demand Cache

The demand cache is suitable when the future resource demand cannot be predicted. The resource environment acquires the resource only when it is needed. This optimizes the cache and achieves a better hit-rate. Once the resource is available, it is stored in the demand cache. All subsequent requests for the resource are satisfied by the demand cache. Once cached, the resource should last long enough to justify the caching cost. Figure 6 illustrates a class diagram for implementing the demand cache [12].

For example, a user can have many roles and one role can have many permissions. Populating the entire permissions domain for all users at the start of an application will unnecessarily overload the cache. The solution is to store the user credentials in the demand cache on a successful log-in. All subsequent authorization requests from the application for already authenticated users will be fulfilled by the demand cache. This way the demand cache will only store a subset of all possible user permissions in the system.

In the absence of a proper eviction policy, the resource will be cached forever. Permanently cached resources will result in memory leak, which degrades the cache performance. For example, as the number of authenticated users grows, the size of the demand cache increases and the performance degrades. One way to avoid this problem is to link resource eviction with resource utilization. In our example, the cache size can be managed by removing the credentials of all logged-out users.

Figure 7: Demand cache example



Predicting the future is a difficult business. In a dynamic environment, adaptive caching strategies represent a powerful solution, based on some sort of application usage heuristics. However, adaptive caching strategies are beyond the scope of this article.

Caching in the ORM World!

Object relational mapping is a way to bridge the impedance mismatch between object-oriented programming (OOP) and relational database management systems (RDBMS). Many commercial and open-source ORM implementations are becoming an integral part of the contemporary distributed architecture. For example, Microsoft Entity Framework and Language Integrated Query (LINQ), Java Data Objects (JDO), TopLink, Hibernate, NHibernate, and iBATIS are all popular ORM implementations. The ORM manager populates the data stored in persistent storages like a database in the form of an object graph. An object graph is a good caching candidate.

The layering principle, based on the explicit separation of responsibilities, is used extensively in the von Neumann architecture to optimize system performance. N-tier application architecture is an example of the layering principle. Similar layering architecture can be used in implementing the ORM caching solution. The ORM cache can be layered into two different categories: the read-only shared cache used across processes, applications, or machines and the updateable write-enabled transactional cache for coordinating the unit of work [13].

Cache layering is prevalent in many ORM solutions; for example, Hibernate's two-level caching architecture [14]. In a layered caching framework, the first layer represents the transactional cache and the second layer is the shared cache designed as a process or clustered cache. Figure 8 illustrates the layered cache architecture.

Transactional Cache

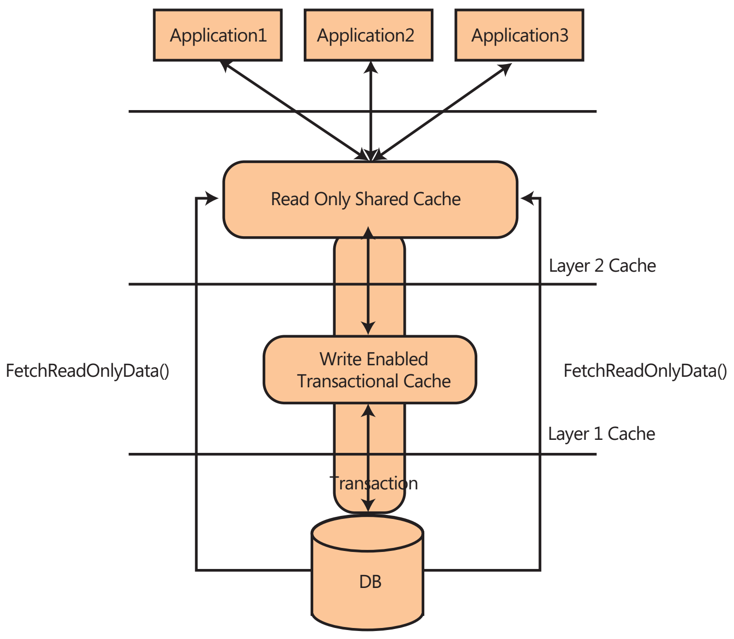
Objects formed in a valid state and participating in a transaction can be stored in the transactional cache. Transactions are characterized by their ACID (Atomicity, Consistency, Isolation, and Durability) properties. Transactional cache demonstrates the same ACID behavior. Transactions are atomic in nature – each transaction will either be committed or rolled back. When a transaction is committed, the associated transactional cache will be updated. If a transaction is rolled back, all participating objects in the transactional cache will be restored to their pretransaction state [15]. You can implement this behavior using the unit of work pattern [13].

Thrashing, cache corruption, and caching conflicts should be strictly avoided in implementing the transactional cache. Many caching implementations offer a prepackaged transactional cache solution, including the TreeCache implementation in JBoss. TreeCache is a tree structured, replicated, transactional cache based on the pessimistic locking scheme [16].

Shared Cache

The shared cache can be implemented as a process cache or clustered cache [14]. A process cache is shared by all concurrently running threads in the same process. A clustered cache is shared by multiple processes on the same machine or by different machines. Distributed caching solutions implement the clustered cache; for example, Velocity is a distributed caching API [1]. The clustered shared cache introduces resource replication overhead. Replication keeps the cache in a consistent state on all the

Figure 8: Layered cache architecture



participating machines. A safe failover mechanism is implemented in the distributed caching platform; in case of a failure, the cached data can be populated from other participating nodes.

Objects stored in the transactional cache are useful in optimizing the transaction. Once the transaction is over, they can be moved into the shared cache. All read-only requests for the same resource can be fulfilled by the shared cache, and since the shared cache is read-only, all cache coherency problems are easily avoided. The shared cache can be effectively implemented as an Identity Map [13]. As shown in Figure 9, requests for the same shared cache resource result in returning the same object.

You can use different coordination techniques to manage the interaction between the shared and transactional cache [15]. These techniques are explained in the following section on intercache interaction.

Managing the Interaction

The interaction between the shared cache and the transactional cache depends on the nature of the cached data. Read-only cached data will result in infrequent cache communication. There are many ways to optimize intercache communication [15].

One solution is to populate the object graph simultaneously in the shared and transactional cache. This saves the overhead of moving objects from one cache to the other. On completion of the transaction, an updated copy of the object in the transactional cache will refresh the shared cache instance of the object. The drawback of this strategy is the possibility of a rarely used transactional cache in the case of frequent read-only operations.

Another solution is to use the just-in-time copy strategy. The object will be moved from the shared cache to the transactional cache at the beginning of a transaction and will be locked for the duration of the transaction. This way no other thread, application or machine can use the locked object. The lock is released on completion of the transaction and the object is moved back to the shared cache (see Figure 10).

It is important to minimize the possibility of a stale or corrupt cache and maximize resource utilization. The data copying between the transactional and shared cache should also be minimized in order to increase the cache hit rate. Since locks are effectively managed in the database, there are some concerns in implementing the same at the application level. This discussion is important but beyond the scope of this article.

Caching domain-specific dependencies is an essential but difficult task. As illustrated in Figure 7, caching the combination of all roles and corresponding permissions for the logged-in user will populate a large object graph. Application patterns like Domain Model and Lazy Load can be effectively applied in caching such domain dependencies [13]. One important consideration in designing a caching strategy is the cache size.

Chasing the Right Size Cache

There is no definite rule regarding the size of the cache. Cache size depends on the available memory and the underlying hardware viz. 32/64 bit and single-core/multicore architecture. An effective caching strategy is based on the Pareto principle (i.e. 80-20 rule). For example, on the ecommerce book portal, 80 percent of the book requests might be related to the top 10,000 books. The application's performance will greatly improve if the list of top 10,000 books is cached. Always remember the principle of diminishing returns and the bell-shaped graph in deciding cache size. (see Figure 11)

How much data should be cached depends on many factors such as processing load patterns, the number of concurrent connections/requests, and the type of application (real time versus batch processing). The goal of any caching strategy is to maximize the application performance and availability.

Conclusion

Small caching efforts can pay huge dividends in terms of performance. Two or more caching strategies and design patterns like GOF [17], PEAA [13], and Pattern of Enterprise Integration (PEI) can be clubbed together to implement a solid caching platform. For example, shared demand cache coupled with a strict time-based eviction policy can be very effective in optimizing the performance of a read-heavy distributed system like the enterprise reporting solution.

Forces like software transactional memory (STM), multicore memory architecture such as NUMA (Non-Uniform Memory Access), SMP (symmetric

Figure 9: Shared cache example

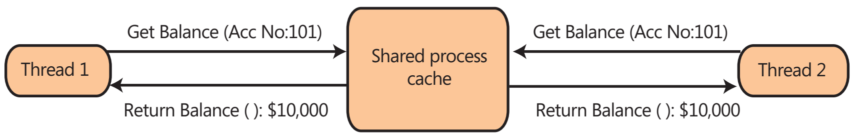
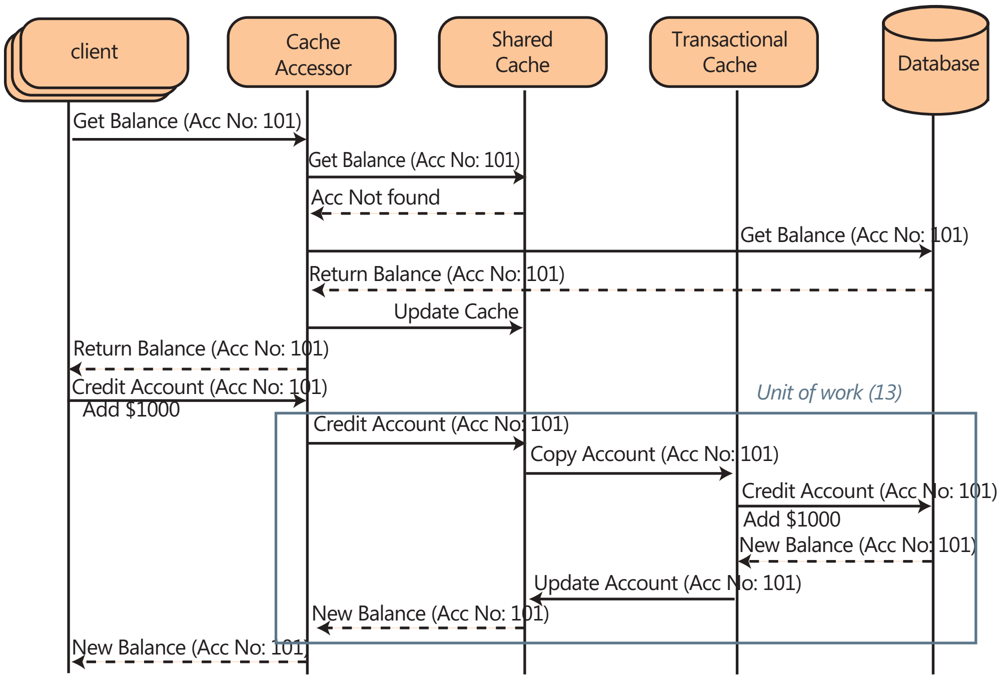


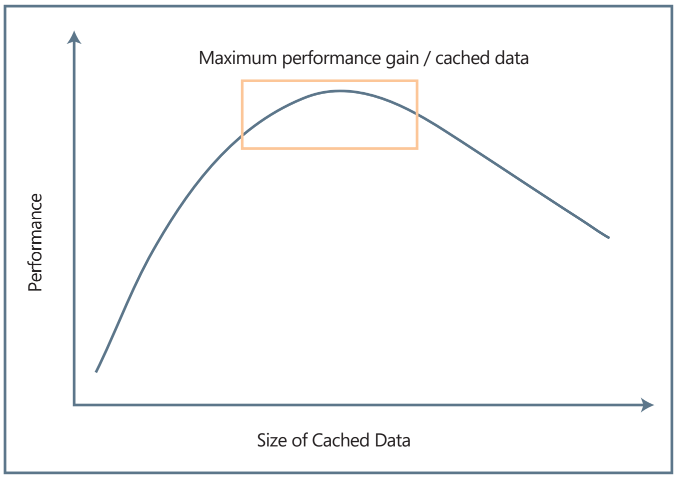
Figure 10 : Intercache interaction


client
Cache Accessor
Shared Cache
Transactional Cache
Database
Get Balance (Acc No: 101)
Get Balance (Acc No: 101)
Acc Not found
Get Balance (Acc No: 101)
Return Balance (Acc No: 101)
Update Cache
Return Balance (Acc No: 101)
Credit Account (Acc No: 101)
Add $1000
Unit of work (13)
Credit Account (Acc No: 101)
Copy Account (Acc No: 101)
Credit Account (Acc No: 101)
Add $1000
New Balance (Acc No: 101)
Update Account (Acc No: 101)
New Balance (Acc No: 101)
New Balance (Acc No: 101)

multiprocessing architectures), and concurrent programming will influence the future of caching platforms. In the era of cloud computing, caching will play a pivotal role in the design of distributed systems. An efficient caching strategy will differentiate a great distributed architecture from the good. Let your next design be a great one.

Acknowledgment
I would like to thank Gita Gadgil for reading the draft material and providing invaluable feedback and suggestions.

Figure 11 : Cached data versus performance



References

1 http://code.msdn.microsoft.com/velocity

2 http://jakarta.apache.org/jcs/getting_started/intro.html

3 http://dictionary.reference.com/browse/cache

4 http://en.wikipedia.org/wiki/Cache

5 Peter J. Denning , "The Locality Principle, Communications of the ACM", July 2005, Vol 48, No 7.

6 http://msdn.microsoft.com/en-us/library/system.web.caching.sqlcachedependency.aspx

7 Michael Kircher and Prashant Jain, "Caching", EuroPloP 2003.

8 Nimrod Megiddo and Dharmendra S. Modha, "Outperforming LRU with an Adaptive Replacement Cache Algorithm," IEEE Computer, April 2004.

9 Kalen Delaney, Inside Microsoft SQL Server 2005 – Query Tuning and Optimization, Microsoft Press, 2007.

10 L.A. Belady, "A study of replacement algorithms for virtual storage computers", IBM Systems J. 5, 2 (1966), 78–101.

11 Octavian Paul Rotaru, "Caching Patterns and Implementation", Leonardo Journal of Sciences LJS: 5:8, January-June 2006.

12 Clifton Nock, Data Access Patterns: Database Interactions in Object-Oriented Applications, Addision-Wesley, 2003.

13 Martin Fowler, Pattern of Enterprise Application Architecture (P of EAA), Addision-Wesley, 2002.

14 Christian Bauer and Gavin King, Java Persistence with Hibernate, Manning Publications, 2006.

15 Michael Keith and Randy Stafford, "Exposing the ORM Cache", ACM Queue, Vol 6, No -3 - May/June 2008.

16 Treecache - http://www.jboss.org/file-access/default/members/jbosscache/freezone/docs/1.4.0/TreeCache/en/html_single/index.html#introduction

17 Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

About the Author
Abhijit Gadkari is an Enterprise Architect with AMG-SIU. Abhijit's background includes an M.S. in IS from Claremont Graduate University, and post graduation in computer science from DOE, India. He has 10 years of software design and development experience. His work and research focuses on building SaaS based IT infrastructure. He blogs on his SaaS related work at http://soaas.blogspot.com.

Distributed Applications in Manufacturing Environments

by Peter Koen and Christian Strömsdörfer

Summary

This article examines some important characteristics of software architecture for modern manufacturing applications. This kind of software generally follows the same trends as other types of software, but often with important modifications. Atypical architectural paradigms are regularly applied in manufacturing systems to address the tough requirements of real time, safety, and determinism imposed by such environments.

Almost all items we use on a daily basis (cars, pencils, toothbrushes), and even much of the food we eat, are products of automated manufacturing processes. Although the processes are quite diverse (brewing beer and assembling a car body are quite different), the computing environments controlling them share many characteristics.

Although we refer to "manufacturing" in this article, we could also use the term "automation," the latter encompassing non-manufacturing environments as well (such as power distribution or building automation). Everything in this article therefore also applies in those environments.

The Automation Pyramid

Software in the industry is often illustrated in a triangle (which is, strangely enough, called the "automation pyramid"; see Figure 1). The triangle shows three areas, with the upper two being rules executed by PC hardware. Although PCs link the world of factory devices with the office and server world of ERP applications, PCs running within the manufacturing application are distinct from those running outside. The manufacturing application works on objects of a raw physical nature (boiling liquids, explosive powders, heavy steel beams), whereas the objects the office and ERP applications handle are purely virtual.

The three levels of the automation pyramid are:

- Information
- Execution
- Control.

The base of the pyramid and the biggest amount of devices in a manufacturing plant are the controllers driving the machinery. They follow the rules and the guidance of an execution runtime, which controls how the individual steps of the process are performed.

Software at the information level controls the manufacturing process. It defines workflows and activities that need to be executed to produce the desired product. This, of course, involves algorithms that try to predict the state of the manufacturing machinery and the supply of resources to optimally schedule execution times. Obviously, the execution of this plan needs to be monitored and evaluated as well. This means that commands are flowing down from information to execution to the control level while reporting data is flowing back up.

The predictability of manufacturing processes and schedules is a tough problem that requires architectural paradigms which may be unfamiliar to developers who deal only with virtual, memory-based objects.

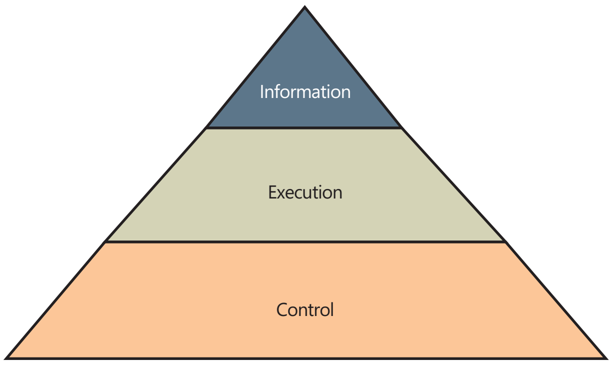
The software discussed in this article lives in the lower part of that triangle, where PCs meet devices, and where the well-known patterns and paradigms for writing successful office, database, or entertainment software no longer apply.

Highly Distributed, Highly Diverse

Most manufacturing tasks involve many steps executed by different devices. To produce a large number of items in a given amount of time, manufacturers employ hundreds or even thousands of devices to execute the necessary steps — very often in parallel. All the intelligent sensors, controllers, and workstations taking part in this effort collaborate and synchronize with each other to churn the required product in a deterministic and reliable way. This is all one huge application, distributed over the factory floor.

The computing systems are rather varied, too. (See Figure 2.) On a factory floor or in a chemical plant, one would probably find all of the following boxes:

Figure 1: The automation pyramid



- Windows Client (everything from Windows 95 to Windows Vista)
- Windows Server (Windows Server 2003 mainly, Windows Server 2008 being introduced
- Windows CE
- Proprietary real-time OS
- Embedded Linux.

All of these systems must communicate effectively with each other in order to collaborate successfully. We therefore regularly meet a full zoo of diverse communication strategies and protocols, such as:

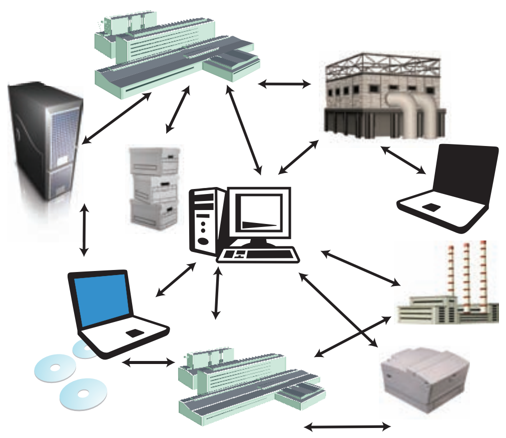
- Windows protocols: DDE, DCOM, MSMQ
- IPC protocols: Pipes, Shared Memory, RPC
- Ethernet protocols: TCP, UDP
- Internet protocols: HTTP, SOAP
- .NET protocols: Remoting, ASP.NET, WCF
- Proprietary real-time protocols.

This diversity of operating systems and means of communication may not look like a system that must be fully deterministic and reliable, but just the opposite is true. However, because of the highly differentiated singular tasks that must be executed to create the final product, each step in this workflow uses the most effective strategy available. If data from a machine tool controller is needed and it happens to be running Windows 95, DCOM is obviously the better choice to access the data than SOAP.

This last example also shows a common feature of factory floors: The software lives as long as the machine that hosts it. Machine tools are expensive and are not replaced just because there is a new embedded version of Windows XP. If a piece of hardware dies, it is replaced with a compatible machine. This may mean that in a 10-year old factory, a Windows 95 machine has to be replaced with a new one that basically has the same functionality, but is based on an OS like Windows CE or Linux. Unless a factory is brand-new, machines from various time periods are present in a single factory or plant and need to seamlessly interoperate.

Because of this diversity, manufacturing applications rely heavily on standards, especially in terms of communication. The variability in operating systems is decreasing somewhat in favor of Windows or Linux solutions and will certainly continue in the future. Also, communication is clearly moving away from proprietary designs toward real-time Ethernet protocols and comparable solutions where applicable. Especially in situations where many different transportation and communication protocols have to work together in one place, the focus for future development is on technologies that can deal with many different protocols, formats, and standards. Windows Communication Foundation is one of the technologies that alleviates a lot of pain in plant repairs, upgrades, and future development. Using WCF, it is possible to integrate newer components into existing communication infrastructures without compromising the internal advances of the new component. There's simply a WCF protocol adapter that translates the in- and outgoing communication to the specific protocol used in the plant. This allows for lower upgrade and repair costs than before.

Figure 2: Distribution



Tough Requirements

Modern manufacturing environments offer a set of tough real-time and concurrency requirements. And we are talking "real-time" here; — time resolutions of 50 µs and less. Manufacturing became a mainly electrical engineering discipline in the first half of the last century, and those requirements were state of the art when software started to take over in the '60s and '70s. The expectations have only risen. For a steel barrel that needs to be stopped, 10 ms is close to an eternity.

Traditionally, these tough requirements have been met by specialized hardware with proprietary operating systems, such as:

- **Programmable Logical Controllers (PLCs)**
  A PLC manages input and output signals of electrical devices and other PLCs attached to it.
- **Numerical Controllers (NCs)**
  A machine tool controller manages axes moving through space (which is described "numerically").
- **Drive Controllers**
  A drive controller manages rotating axes (such as those of a motor).

These extreme requirements are not expected from PC systems yet, but they, too, must react to very different scenarios in due time. And usually there is no second try. A typical task executed by a PC on a factory floor is to gather messages coming from PLCs. If an error occurs in one of them, it regularly happens that all the other PLCs also start sending messages, like a bunch of confused animals. If the PC in charge misses one of those messages (for instance, because of a buffer overflow), the message is probably gone for good, the cause of the failure might never be detected, and the manufacturer must subsequently throw away all the beer that was brewed on that day.

To summarize:
- Hard real-time tasks (measured in microseconds) rely mainly on specialized hardware and software.
- PC systems must still meet "soft" real-time requirements (microseconds become milliseconds).
- In manufacturing applications, there is no undo and no redo.

Real-World Parallelism

Factories are rarely built to manufacture a single product synchronously. Rather, they try to make as many copies of the product concurrently as possible. This induces massive parallelism, with appropriate heavyweight synchronization. Today this is implemented as a kind of task parallelism, with each computing unit running as a synchronous system taking care of one of those tasks. All parallelization moves into the communication. (More on how this is embedded in the overall software architecture in the section on architectural paradigms.)

With the advent of multicore chips, however, the current patterns and paradigms need to be revisited. It is still unclear what parallel execution within one of the aforementioned controllers could mean or whether parallelism makes sense within the current execution patterns.

On PC systems, however, multicore architectures can readily be applied for the typical tasks of PCs in manufacturing applications such as data retrieval, data management, data visualization, or workflow control.

- Parallelism in manufacturing applications appears today mainly in the form of asynchronous communication.
- Multicore architectures have a yet-to-be-determined effect in real-time controller systems.

PCs in manufacturing, however, typically carry out tasks that can take advantage of multicore architectures right away.

Security + Safety + Determinism = Dependability

Although security and safety are extremely important almost everywhere, manufacturing always adds life-threatening features such as exploding oil refineries or failing breaks in cars. If a computer virus sneaks into a chemical plant undetected, everything and anything might happen, not the worst of which were depredating the company's accounts.

It is better to use the term "dependability" here, because this is what it is all about. Both the manufacturer and the customers consuming the manufacturer's products depend on the correct execution of the production. This comes down to the following features:

- **Security**
  Manufacturing applications must not be exposed to the outside world (i.e. the Internet) such that anyone could penetrate the system.
- **Safety**
  The application must be fail-safe; even in the unlikely event of a software or hardware error, execution must continue, e.g. on a redundant system.
- **Determinism**
  The execution of a manufacturing application must be deterministic, 24/7. This does not mean that fuzzy algorithms could not be applied somehow (for instance, to optimize manufacturing with a batch size of '1'), but even that must be reliably executed within a certain time period.

The security requirements make it more or less impossible to connect a factory to the Internet or even to the office. Contact is therefore made based on three principles:

- The communication flow into and out of the factory floor is never direct, but relayed through proxies and gateways.
- Communication is usually not permanent; special tasks such as backup of production data, rollout of software updates, and so on are applied in dedicated time spans.
- Many tasks have to be executed physically inside the factory (by wiring a secure portable computer directly with the plant for maintenance purposes, for instance).

"The software controlling these objects cannot rely on transactional or load-balancing patterns because there simply is no rollback and no room for postponing an action. "

Architectural Paradigms

As already mentioned, manufacturing applications do not control virtual objects such as accounts or contracts, but real physical objects such as boiling liquids, explosive powders, or heavy steel beams.

The software controlling these objects cannot rely on transactional or load-balancing patterns because there simply is no rollback and no room for postponing an action.

This results in the application of a rather different architectural paradigm in those environments alongside the more traditional event-driven mechanisms: using a cyclic execution model.

Cyclic execution means that the operating system executes the same sequence of computing blocks within a defined period of time over and over again. The amount of time may vary from sequence to sequence, but the compiler and the operating system guarantee that the total time for execution of all blocks always fits into the length of the cycle.

Events and data in the cyclic paradigms are "pulled" rather than "pushed," but the "pull" does not happen whenever the software feels like it (which is the traditional "pull" paradigm in some XML parsers), but rather periodically. This is the main reason why the term "cyclic" is used instead of "pull" (or "polling").

The cyclic execution model reduces complexity:

- In a cyclic system, the number of states the system can adopt corresponds directly to the product of all variable states. In an event-driven system, the sequence of changes and variables has to be taken into account as well, which makes the number of states the system can adopt completely unpredictable.
- Event-driven models operate based on changes between variable states, whereas the cyclic model works on states of variables. The rules of combinatorial algebra teach us that the number of transitions from one state to another is much larger than the number of states themselves. *This is particularly relevant in distributed systems, where the connection between nodes is potentially unreliable.*
- The difference between minimal and maximal load is considerably lower in a cyclic system: The minimal load is the computing time needed to run the cyclic itself, the maximal load is 100 percent CPU time for a whole cycle. An event-driven system is more "efficient" in that it only does something if it needs to. On the other hand, with a lot of events happening more or less simultaneously, the maximal load is rather unpredictable.

Of course, cyclic execution has a number of drawbacks, too. It uses CPU power even if there is nothing to do. It also limits the duration of any task by the duration time of the cycle, which is more likely to range between 100 µs and 100 ms than a second or more. No CPU can do a lot of processing within 100 µs.

A system cannot rely on cyclic execution only. In any environment, things might happen that cannot wait until the cyclic system fancies to read the appropriate variables to find out something's just about to blow up. So in every cyclic system there is an event-driven demon waiting for the really important stuff to kill the cycle and take over. The problem here is not whether this kind of functionality is needed or to what degree, but rather how to marry these two approaches on a single box in a way that allows both operating systems to work 100 percent reliably.

The question of how to deal with cyclic and event-driven patterns within a single system becomes highly important to PCs that are part of a distributed manufacturing application. Here lives a system dominated by an event-driven operating system, communicating with mainly cycle-driven systems. Depending on the job, things must be implemented in kernel or user mode.

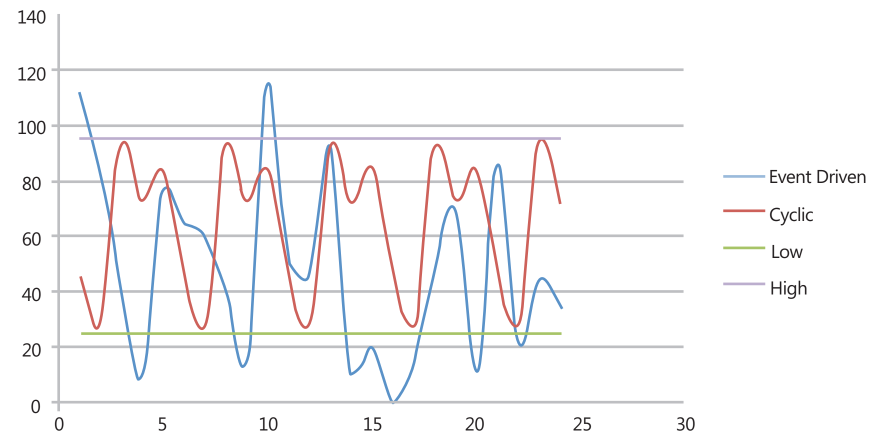
- Collecting data from devices on the factory floor is done in user mode by mimicking the appropriate cycles with timers and reading and writing the variables after each cycle.
- Alarm messages, on the other hand, are often mapped to a specific interrupt, such that a kernel extension (not just a driver, but a sort of extended HAL) can acknowledge the message as soon as possible and perhaps spawn a real-time operating system that sits beside Windows on the same box.

Those are just two examples of how architects try to let the two paradigms collaborate. None of them works extraordinarily well, although the known solutions are all robust enough today for 24/7 factories. What makes them work, in the end, is probably the huge amount of testing and calibration of timers, threading, and so on that goes into the final products. This shows clearly that there is plenty of room for improvement.

A lot of hope for future improvements rests on the principles of Service Oriented Architecture. By allowing cyclic and event-driven systems to be loosely coupled through messages, it is possible to more efficiently route information in the system depending on severity of the event.

Cyclic systems could then do what they are best at – delivering predictable resource usage and results and simply use a specific time in the cycle to put relevant information into a message queue which then would be serviced by other systems, cyclic or event driven. This approach would also lessen the chance of error from pulling data from a specific location/variable by queuing the necessary data.

Figure 3: Architectural paradigms – resource usage with cyclic versus event-driven architecture



The Future of Manufacturing Applications

The architectural trends in manufacturing software can be summarized as follows:

- Standard operating systems are applied wherever possible. The more configuration capabilities a particular system offers, the more it will be used.

This development is intensified by the current trends in virtualization. Since manufacturing environments typically involve the duality of device operation by a real-time operating system and visualization and manual control by a Windows system, the offer of having both available on a single box is extremely intriguing for technical and financial reasons.

- Standard communication protocols and buses are applied wherever possible. The transmission and processing time for a protocol remain most critical, however. Internet protocols based on HTTP and XML will therefore probably never be used widely.
- The Internet is entering the world of factories very slowly, albeit mostly in the form of an intranet. This means in particular that Internet technologies are already very important, but rather as a pattern than in their original form. Software updates are rolled out as "Click Once" applications, but via intranets and disconnected from the Internet.

This certainly also applies to cloud computing. The offer to have a distributed environment look like a single entity is too tempting not to exercise it. The mode, however, will differ from the typical cloud application. Factories employ their own "local" cloud than "the" cloud.

Currently, there is a lot of discussion and rethinking going on in the manufacturing space. What will the future look like? Should the development favor more cyclic or event-driven execution?

Multicore CPUs add an interesting angle to this discussion. On the one hand, it would mean that there is enough power to simply make everything cycle based and have several cycles run in parallel. On the other hand, it would favor event-driven systems since the workload can be easily distributed.

We think that the answer is somewhere in the middle. The most interesting scenarios are enabled through the implementation of SOA principles. By changing many parts of the solution into services, it is easier to host cyclic and event-driven modules on the same machine or on several devices and have them interact. Manufacturing systems are applying modern technology and architecture paradigms to automatically distribute load and move operations between various nodes of the system depending on availability of resources. When you think about it, manufacturing is very similar to large-scale Internet applications. It's a system of various, similar, but still a little bit different devices and subsystems that operate together to fulfil the needs of a business process. The user doesn't care about the details of one specific device or service. The highest priority and ultimate result of a well-working manufacturing system is the synchronization and orchestration of devices to build a well-synchronized business process execution engine that results in a flawless product. It's not much different from the principles you can see today with cloud computing, web services, infrastructure in the cloud, and Software as a Service applications.

Taking analogies of enterprise computing to other areas of manufacturing execution and control opens up a whole new set of technologies, like virtualization and high-performance computing. Virtualization would allow moving, replicating, and scaling the solution in a fast way that simply isn't possible today. For computation-intensive tasks, work items could be "outsourced" to Windows Compute Cluster Server computation farms and wouldn't impact the local system anymore. Allowing for more cycles or respectively for handling more events has a dramatic impact on the architectural paradigms in manufacturing, and no one can say with 100-percent certainty what the model of the future will be. On-demand computational resources could very well enable more use of event-driven architectures in manufacturing.

While these technical advances sound tempting, there are still a lot of problems that need to be solved. For example, there's a need for real-time operating systems and Windows to coexist on the same device. This can't be done with current virtualization technology because there's currently no implementation that supports guaranteed interrupt delivery for real time OSes.

Cloud Computing and Manufacturing

When you compare the cloud offerings from multiple vendors to the needs in manufacturing, cloud computing looks like the perfect solution to many problems: integrated, claim-based authentication, "unlimited" data storage, communication services, message relaying, and the one factor that has changed manufacturing and software development as no other factor could have done — economics of scale.

Unfortunately, there is one basic attribute of cloud services that make the use of current cloud offerings in manufacturing pretty much impossible: the cloud itself.

Manufacturing is a highly sensitive process and any disruption could not only cost millions of dollars, but also be a serious risk to the lives of thousands of consumers. Just imagine what would happen if a hacker got access to the process that controls the recipes for production of food items. It is pretty much unthinkable to ever have a manufacturing environment connected to the cloud.

Still, plenty can be learned from cloud-based infrastructure and services. Let's look at SQL Server Data Services as an example. SSDS is not simply a hosted SQL Server in the cloud. It's a completely new structured data storage service in the cloud; designed for simple entities that are grouped together in containers and a very simple query, which allows direct access to these entities. The big advantage of SSDS is that it is a cloud-based solution that hides the details of replication, backup, or any other maintenance task from the application using it. By providing geo replication and using the huge data centers from Microsoft, it can guarantee virtually unlimited data storage and indestructible data.

Hosting a system like this on premise in manufacturing plants would solve many problems we are facing today: Manufacturing processes are generating a vast amount of data. Every single step of the production process needs to be documented. This means we are looking at large amounts of very simple data entities. The data collection engine cannot miss any of the generated data; otherwise, the product might not be certified to sell it. And one very interesting aspect of manufacturing systems is that once a device starts sending an unusually high amount of data, for example due to an error condition, this behavior quickly replicates on the whole system and suddenly all devices are sending large volumes of log data. This is a situation that's very difficult to solve with classical enterprise architecture database servers.

The automated replication feature is also of great use: Many manufacturing systems span thousands of devices on literally thousands of square feet of factory floors – many times even across buildings or locations. Putting data in time close to where it is needed is essential to guarantee a well-synchronized process.

While cloud computing is still in its infancy and almost all vendors are busy building up their first generation of services in the cloud, the applicability of those services as a on-premise solution is often neglected to stay focused on one core competency.

About the Authors

Christian Strömsdörfer is a senior software architect in a research department of Siemens Automation, where he is currently responsible for Microsoft technologies in automation systems. He has 18 years of experience as a software developer, architect, and project manager in various automation technologies (numerical, logical, and drive controls) and specializes in PC-based high-performance software. He received his M.A. in linguistics and mathematics from Munich University in 1992. He resides in Nuremberg, Germany, with his wife and three children.

Peter Koen is a Senior Technical Evangelist with the Global Partner Team at Microsoft. He works on new concepts, ideas, and architectures with global independent software vendors (ISVs). Siemens is his largest partner where he works with Industry Automation, Manufacturing Execution Systems, Building Technologies, Energy, Healthcare, and Corporate Technologies. Peter resides in Vienna, Austria and enjoys playing the piano as a way to establish his work/life balance.

Upcoming Events

MS Professional Developers Conference (PDC08)
October 27-30, 2008, Los Angeles, Calif.
http://microsoft.com/pdc

patterns & practices Summit
November 3-7 2008, Redmond, Wash.
http://www.pnpsummit.com/west2008/west2008.aspx

TechEd EMEA IT Pros
November 3-7 2008, Barcelona, Spain
http://www.microsoft.com/emea/teched2008/itpro/

TechEd EMEA Developers
November 10-14, 2008, Barcelona, Spain
http://www.microsoft.com/emea/teched2008/developer

DevConnections
November 10-13, Las Vegas, Nev.
http://www.devconnections.com/

InfoQ Qcon
November 17-21, 2008, San Francisco, Calif.
http://qconsf.com/sf2008

IT Virtualization Live
December 9-11, 2008
http://itvirtualizationlive.com/

Next Issue on Green Computing – Subscribe FOR FREE
The ubiquity of technology services is leading to significant energy waste, in a world where increasing energy costs and the need to reduce carbon footprints are becoming an imperative. Learn about practitioner experiences and ideas on sustainable IT environments and solutions.
Go to **http://www.architecturejournal.net**
and subscribe at no charge.

Architecture Guidance Is Coming Back
MS patterns & practices is working on the next version of its Application Architecture Guide. Solution architects, developer leads and developers can find here a fresh repository of principle-based, durable, and evolvable guidance.
Stay tuned on this work-in progress effort between patterns & practices, product teams, and industry experts.
http://www.codeplex.com/AppArch

Platform
Architecture Team

Microsoft®

17

subscribe at
www.architecturejournal.net

098-110484