

Openwings

A Service-Oriented Component Architecture for Self-Forming, Self-Healing, Network-Centric Systems (Rev 2.0)



by

Guy Bieber, Lead Architect, ISD C4I and

Jeff Carpenter, Software Engineer, ISD C4I



ABSTRACT - *The computer revolution is over. Computers are everywhere, and are becoming increasingly invisible to the average consumer. The network revolution has just begun: our desktop computers are networked, as well as our cell phones, PDAs, and more and more devices in our homes. Current software architectures do not provide the consumer with the most benefit from this connectivity; it is very difficult for people to set up, manage, and interact with networked devices around them. Networked systems today are static and brittle; they usually consist of a collection of stovepipe client-server applications where the clients only communicate with particular servers, with particular protocols, in very specific environments. The Openwings architecture incorporates the latest commercial technologies and concepts to provide a service-oriented architecture where components spontaneously publish and discover services, independent of transport protocols and deployment environments. This architecture will allow the next generation of software applications to be reusable, interoperable, available, and to exploit spontaneous networking, allowing people to interact with their environment and other people in much richer ways.*

KEY WORDS – *Architecture, Component, Discovery, Java, Jini, Network Appliance, Network-centric, Openwings, PAN, Self-Forming, Self Healing, Service-Oriented*

THE NETWORK REVOLUTION

The tremendous growth in network bandwidth has driven the convergence of voice, video, and data to IP based networks. The bandwidth available on wired networks is increasing by a factor of 10 every two years. Wireless bandwidth is doubling every 9 months [1]. Technologies such as Gigabit Ethernet, Wireless IP [2], and Bluetooth [3] are making bandwidth more abundant and inexpensive. Meanwhile, computer processing speed, memory, and disk space doubles every 18 months. Simultaneously, the size of computing platforms continues to decrease. Today's handheld PCs are often more powerful than desktop computers of the 90s. Once a rare commodity, computers are now embedded in everything - toys, cars, cell phones, even breadmakers.

The Network Revolution

- Network bandwidth growth will drive the convergence of voice, video, and data.
- The computers embedded in all consumer devices will have a digital presence on the network.
- Personal Area Networks (PAN) will provide proximity based spontaneous networking.
- Through spontaneous service discovery, software will be able to interact in new and powerful ways.

When these embedded computing devices are networked, tremendous possibilities arise. In the future, almost every device imaginable will have a digital presence on the network [4]. Metcalfe's Law states that the power of a network is proportional to the number of nodes on it squared. This is true for the same reason that people can achieve amazing feats when they work together; witness the pyramids, the Space Shuttle, and even the shopping mall. When computers work together, they enrich the interaction of people with their environment and other people. The time is coming when all kinds of hardware and software will integrate seamlessly and become a natural part of a person's environment. Today's "born to surf" generation will give way to the "born in the web" generation.

Current software architectures are not up to the challenge of a highly networked, dynamically changing environment. The client / server model prevalent today is too static and brittle. Typically, clients only communicate with the servers for which they were originally written, using specific protocols, under specific deployment conditions. The interface between the client and server is private, leading to a collection of closed stovepipe systems. Most of these implementations also assume the underlying network is static and reliable. The future of networking breaks all of these assumptions. With the advent of Personal Area Networks, such as Bluetooth, the network is very dynamic. As a person moves, the set of devices visible to them changes. There is also an increasing use of mobile hosts that dynamically move into and out of networks [5, 6].

The network revolution has left software architecture lagging behind. Service-oriented, discovery-based component architectures help the network revolution realize its full potential in both commercial and military applications.

THE COMMERCIAL USE CASE

In a world where most devices have networked computers embedded in them, people will be able to interact with their environment in much richer ways. This applies to all aspects of daily life, in the home, office, or car – anywhere there is a network. These network appliances will require no configuration or loading of drivers. Instead, they are simply powered on and attached to the network, either by a physical connection or by bringing them into the proximity of a wireless network. These devices are then incorporated into a personal space for use. These appliances will deliver services to the user through whatever interface device the user possesses, such as a phone, television, Personal Digital Assistant (PDA), or microphone. Here are some examples of what will be possible in a networked, service-oriented world:

- Music files and playlists stored at home are a service that can be accessed in the car, at the gym, or at work.
- A person will be able to walk into any room and view a list of the services in that room on a PDA.
- PDAs will automatically synchronize data with personal computers when they are co-located [7].
- A person’s phone always has their latest list of contacts, because it talks to their home service context to get them.
- When a person arrives at home, the door automatically unlocks, the entryway lights come on, and the heat is adjusted to their preset preference [7].
- Nobody will wait in line at the movies again. Instead, people will purchase tickets from a cell phone or PDA as they walk into the theater [7].
- People will be able to query their environment for services or information. At the grocery store simply saying, “Where’s the Italian olives?” into the phone will display a digital map to their location.
- Computers, stereos, and televisions will interact to direct video, audio, or data anywhere in the house.

The net result is that people will interact with their environment more efficiently and with much greater flexibility. Appliances will be able to do more valuable things when they are put together. However, with all of this personal information available on the network, security and privacy will be very important to the consumer. The online industry has learned this the hard way in several well-publicized cases, such as the incident where thousands of

credit card numbers were stolen from an online music retailer and placed on the Internet.

For consumers, the network revolution holds both the promise of powerful functionality and the increased risk of privacy invasion.

THE MILITARY USE CASE

The military presents one of the most dynamic and hostile environments for system deployment. System elements are continually moving, reorganizing, appearing, and disappearing, and the enemy is waging a physical and electronic information battle to destroy these systems. In this environment, availability and security are essential. The Department of Defense (DoD) has established a vision for Command Control Communications Computers and Intelligence (C4I) systems of the future: Joint Vision 2010. This vision is of a network-centric battlefield, where all battlefield nodes are on a network that is highly mobile and adaptive [8].

C4I systems used today fall short of the vision; they act as a collection of stovepipes and only collaborate in very primitive ways. The DoD continues to look to commercial markets for solutions to its architecture problems. The DoD has identified a common set of problems from various warfighting exercises and experiments, as seen in the chart below. Current military systems are an administration nightmare. Manually configuring the hundreds of networks and thousands of devices in a typical Tactical Operations Center (TOC) is extremely costly and difficult. These systems are also very difficult to assemble and disassemble, preventing the mobility so critical to military operations.



Figure 1. DoD Architecture Issues

There are three classes of battlefield elements: collection systems (sensors), engagement systems (shooters), and command and control (C2) nodes. The battlefield of the future will tie these elements together on a Global Battlefield Information Network (GBIN). Future systems will use this network to collect, process, and disseminate information to decision makers anywhere on the battlefield.

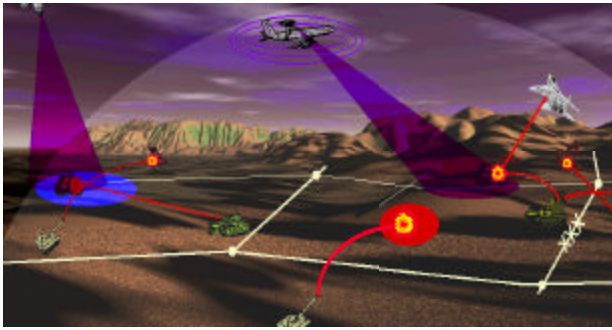


Figure 2. Dynamic Battlefield Environment

The Tactical Operations Center (TOC) is an example of a C2 element. The current TOC is a collection of systems that provide various isolated views of the battlefield. The equipment is large, difficult to configure, requires significant administration, and is very immobile. The majority of communication between the systems consists of custom text messages.

To reduce the administration of the TOCs, the formation of systems and systems of systems has to be simplified. Adding a printer or radio to the network should be as simple as plugging it in.

By making services in the TOC discoverable, they can be shared between systems, reducing the overall size of the TOC. For example, large Electronic Maps are used throughout the TOC, and sharing them saves disk space on each system. Hardware components such as radios and printers can also be shared throughout the TOC, reducing both the size of individual shelters and the number of shelters needed to form a TOC.

Currently, communication between systems in the TOC requires tremendous amounts of manual configuration. If the services of one system could be automatically discovered by another system and utilized, it would dramatically reduce system setup time.

In a battlefield situation equipment often goes down or is destroyed. In these cases the ability to discover replacement services in the environment is critical to availability.

SERVICE-ORIENTED ARCHITECTURE

The client-server architectures of the last decade are simply not flexible or scalable enough to build consumer or military solutions in this new era of networking. Instead, the industry is looking to service-oriented architectures to solve design problems. This section introduces the concepts and terminology of service-oriented architecture, also shown in figure 3.

- **Component** - Components are individually deployable, binary implementations that provide contractually specified services. Components are subject to 3rd party composition and deployment [9].

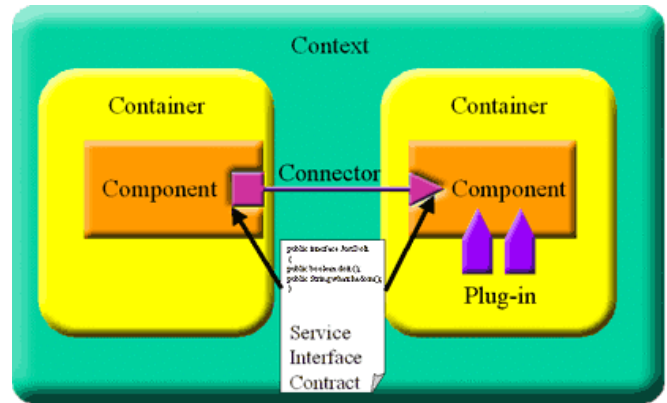


Figure 3. Service-Oriented Programming Elements

- **Connector** – Connectors represent interactions between components. A connector allows two components to communicate using a specific protocol. Each side of a connector represents a specific role and plugs into a port of a component. Connectors can be created for synchronous, RPC style protocols and asynchronous, message style protocols.
- **Container** - Containers hold components and manage their lifecycle. Examples of containers in other architectures include EJB Containers (contain EJBs), Web Servers (contain servlets), and Web Browsers (contain applets).
- **Contract** – Contracts defined using java interfaces by specifying syntax and semantic behavior. Contracts represent services and are the core unit of interoperability in Service-Oriented Programming (SOP).
- **Context** – Contexts provide a deployment environment for components. This environment prescribes policies regarding to security, discovery, and system boundaries. Relationships can be formed between contexts to have services shared across contexts.
- **Discovery** - Discovery allows Service-Oriented applications to dynamically find and publish services. Discovery is a core concept that allows Service-Oriented applications to adapt to their environment.

Service-Oriented architectures make it possible to create reusable software components. A shared vision of the software industry is a marketplace of such components, where building a new system is as easy as buying components and connecting them together. This vision carries the promise of little or no coding, and dramatically increased productivity. However, there several factors that are preventing the industry from achieving this vision:

- The industry hasn't been able to agree on how to connect components together. There are hundreds of middleware products and standards.

- Existing software has too much information embedded in it about the environment it is deployed in. This prevents the software from being used in other external contexts.

A component model that is independent of protocols and deployment environment is essential to further the software component market.

DISCOVERY AND SERVICE LOOKUP

A key issue in a service-oriented architecture is how components locate services. Configuration of software and hardware components has typically been treated as a static operation. Service location information is often hard coded in software components, or stored in a configuration file that the component reads on startup.

In reality, no networked system is completely static. Nodes enter and leave the network, software and hardware components are replaced or upgraded. If the rate of change in the system is slow enough, it is possible to keep the system running with a static configuration model. As the rate of change in a system increases, the expertise of system administrators becomes more and more critical to the successful operation of the system. Most systems are becoming more and more dynamic, which is creating a huge management problem for statically configured systems. These systems are simply not built to recover from network faults or failed services.

The solution to this problem consists of two concepts: discovery and service lookup. Instead of static configuration data, components “discover” the environment in which they are deployed and “lookup” services they need.

Network based discovery deals with how computers network themselves together. The most commonly used technology in this area is Dynamic Host Configuration Protocol (DHCP), an IP based protocol that allows a node to discover the network and obtain an IP address. The latest discovery technologies are based on spontaneous, proximity-based wireless networking, such as Bluetooth [3] and Wireless IP [2].

Discovery and lookup for software components is an emerging area. Naming services such as Lightweight Directory Access Protocol (LDAP) and Java Naming and Directory Interface (JNDI) have been around for a while. These services allow components to lookup services by name, but instead of discovery, they require static configuration of the name service. The scalability of the system is limited, since components must perform lookups by name. Components need a richer set of semantics for locating services.

There are several different technologies that are targeted at service-oriented architectures, including Sun’s Jini Technology [10], Microsoft’s .NET[12], JXTA [27], and Personal Area Networks (like Bluetooth). Each of these

technologies is targeted toward a specific type of discovery as shown in the following figure.

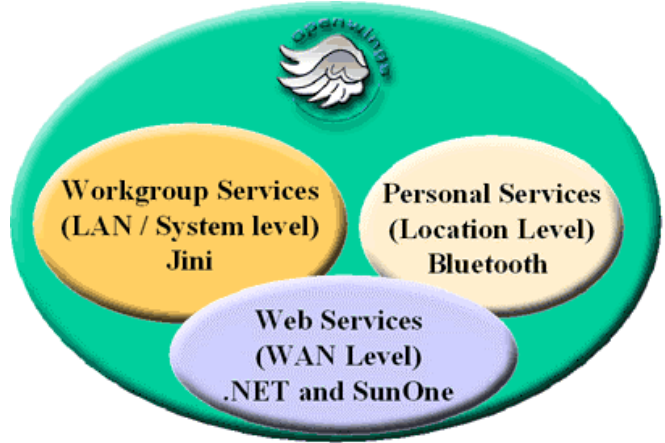


Figure 4. Discovery Mechanisms

Of the existing service-oriented technologies, Jini is the most mature, and it provides the best semantics for service-oriented discovery and lookup. Jini is based on the Java language and its Remote Method Invocation (RMI) model. In the Jini architecture, the repository of services is called a “lookup service”. Components discover lookup services and use them to advertise services they provide (referred to as “join”), along with additional attributes that characterize the service. Service lookup is based on the service interface, a unique identifier, service attribute matching, or any combination of these. Jini discovery consists of a combination of directed and/or broadcast network traffic, allowing components to locate lookup services in a variety of different network topologies. Broadcast style messages are used to discover lookup services in the local area network, while directed messages are used to discover lookup services on a wide area network.

All of these discovery models have common underlying Service-Oriented elements. Openwings brings these models together by allowing discovery mechanisms to be added to the framework as plug-ins. This allows web, workgroup, and personal services to be provided and used transparently by Openwings components. By enabling interoperability between these models, Openwings fulfills the vision of a Service-Oriented Architecture.

OPENWINGS ARCHITECTURE OVERVIEW

The goal of the Openwings architecture is to provide a Service-Oriented Programming (SOP) framework for highly dynamic networked systems of software and hardware components. The focus of the architecture is components, which use and provide discoverable services. Every component is a manageable, reusable encapsulation of functionality that can be dropped into any context because they are independent of platform, protocol, environment and database. Systems are self-forming, self-healing and highly available.

The Openwings architecture has been designed based on best-practice principles drawn from a variety of new technologies and industry-wide initiatives, with dependencies on only one core technology: Java. Openwings is independent of practically all other technologies, by providing plug-ins and abstractions of these technologies.

The architecture will be used in a variety of ways during the development of a product or system:

- System engineers use the architecture to design and prototype component-based systems
- Software developers use the architecture to develop reusable components
- System managers use the architecture to get a view of the status of a deployed system
- Users will experience true hardware/software plug and play and new levels of interoperability between devices they use.

The core architecture consists of seven major service categories: Component Services, Connector Services, Container Services, Context Services, Security Services,

The Openwings Vision

- Utilize best industry standards, technology, and practices.
- Independence from environments, platforms, protocols, and databases
- Everything is a component
- All components are reusable and interoperable
- Network formation and service discovery is dynamic.
- Systems are Self-Forming and Self-healing
- Availability and Security are built into the architecture
- Ease of use – Zero administration
- Ease of development – use and provide services

Management Services, and Install Services. The following figure shows high-level view of the core Openwings architecture:

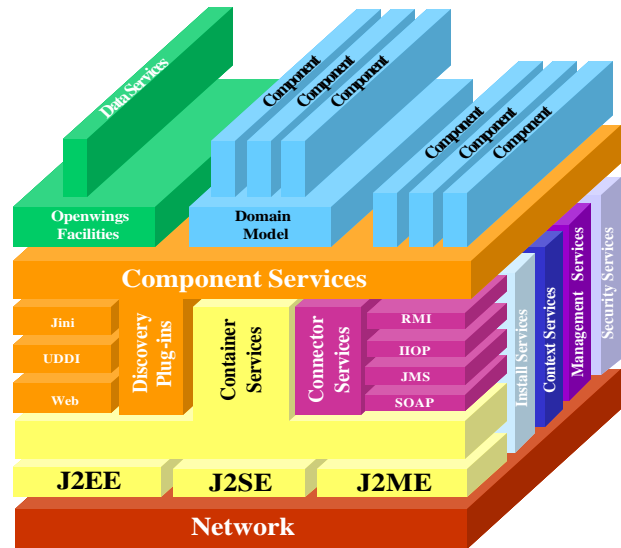


Figure 5. Openwings Architecture

- Component Services encapsulate the details of providing and using services.
- Connector Services hide the details of inter-process communication and allow applications to be independent of transports.
- Container Services provide for life cycle management of components, enforce code security, and provide high availability features.
- Context Services provide information about the deployment environment and enforce policies regarding to system level boundaries.
- Install Services provide for installation of components, application serving of components, resolution of dependencies, and resolution of policies.
- Management Services provide policy-based management to automatically administer systems.
- Security Services provide authentication, authorization and privacy.

On top of Openwings, components, domain models, and facilities are defined. Data Services is an Openwings facility that provides an object based three-tier database model. The Openwings architecture is designed to be applicable to almost any problem domain, with the exception of hard real time systems. The first domain model under development is for the military use case. It is a Command, Control, Communications, Computers, and Intelligence (C4I) framework called FreedomC4I.

COMPONENT SERVICES

Component Services provides a Service-Oriented Programming (SOP) abstraction of service discovery and lookup. Component Services also encapsulates the details of using Connector Services. In Openwings, every self-contained unit of functionality treated as a component, whether it is hardware or software based. Hardware components are wrapped in software.

Component Services abstracts the details of discovery and lookup so that components can easily communicate, whether they are in the same Local Area Network (LAN), connected by a Wide Area Network (WAN), or even running in the same Java process. Component developers are only concerned with a small set of operations. For synchronous services this means providing and using services, removing a provided service, and discarding a used service. For asynchronous services this means publishing and subscribing to services, unsubscribing a service, and unpublishing a service.

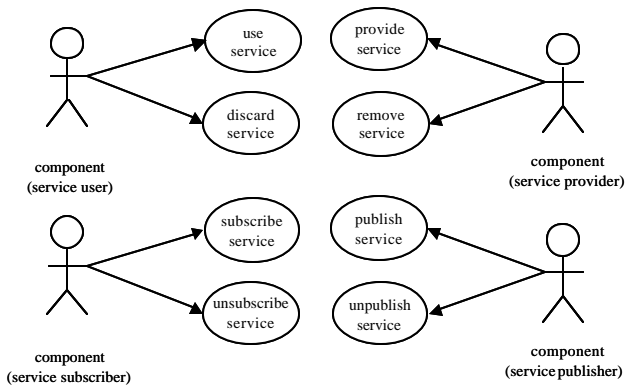


Figure 6. Component Services Use Cases

These use cases provide a vastly simplified programming model that is ideal for the constraints of smaller devices. The code is as simple as obtaining an object representing the component from a factory and calling “useService()” or “provideService()”. Even the details of the protocol used for the service are handled transparently through the use of Connector Services. Component Services provides several variations on these basic use cases, adding a richer set of functionality not provided by existing component models and discovery mechanisms.

In an adaptable system, some components will need to use services without prior knowledge of the service interface. Component Services provides the ability to attach user interface adapters to services so end users can access services without previously installed software. An interface adapter could be a panel for a certain display, a speech-based interface, or a web page. Any number of user interface adapters can be created for a given service and deployed in different environments. This design is based on the Model-View-Controller design pattern and provides a clean

Component Services makes the dream of reusable Components reality

- Simplified programming model for service-oriented computing
- Hides details of discovery
- Connector Services is used to abstract connector details
- Attach a locale-specific user interface adapter to any service
 - Isolate environment specific details and develop reusable business logic

separation of the functionality of services and user interfaces. This has the added benefit of simplifying internationalization of interfaces. The design integrates with related efforts such as the Jini Community ServiceUI framework [15] and Java Server Pages [16], a technology for creating dynamic web content.

Component Services utilizes the Openwings Policy and Management Services to provide environment independence, so that component developers can focus on business logic. Component developers can isolate environment-specific or other configurable information in policies and provide default settings in the implementation. Component Services also provides an extensible Component Management framework for dynamic control of policies and component management beans, which provide pluggable management functionality. Component Services provides a default set of management beans and a framework for adding custom management beans to a specific component or system.

CONNECTOR SERVICES

Connector Services provides transport protocol independence by abstracting synchronous and asynchronous communications in terms of Java interfaces. Interchangeable connectors handle the details of specific protocols. The connector framework is designed so that it can be used independently of the rest of the architecture in other frameworks, such as Enterprise Java Beans (EJB). Connector Services specifies an API for connectors and facilities for the generation and location of protocol specific connectors. Example synchronous protocols are Java RMI, CORBA, DCOM and (Remote Procedure Call (RPC). Example asynchronous protocols are Java Messaging Service (JMS) [17], and the CORBA Message Service (CMS) [18].

The following Architecture Description Language (ADL) diagram [19] shows the anatomy of a connector. A connector is composed of a user proxy and a provider proxy. A connector is installed when each proxy is inserted in the appropriate component.

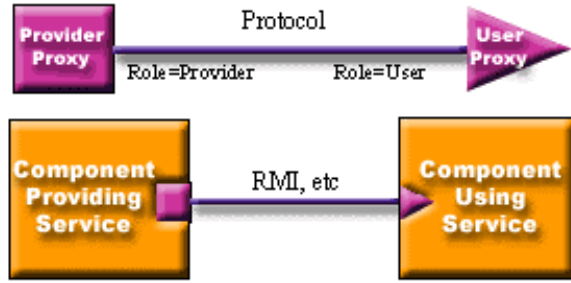


Figure 7. Connector Architecture

Connector Services include a set of tools that can be used to create and load connectors. Protocol-specific Connector Generators generate connector source code based on service interfaces created by component developers. Interfaces must be written in the Java programming language. Every method of a synchronous interface must throw `java.rmi.RemoteException`, since these methods can fail in ways that local method calls cannot [20]. Asynchronous interfaces must contain methods with no return values.

Connector Compilers are used to convert connector source code into a deployable format, such as a Jar file. A Connector Loader is used to instantiate a connector based on the location of the deployed connector.

The Openwings Connector architecture also allows for the easy integration of legacy applications. By creating half of a connector that talks back to a legacy service programs potentially written in other languages can be integrated.

This design is meant to encourage middleware vendors to extend the connector infrastructure by providing their own protocol-specific connector generators. Connector Services provides flexibility to allow interchangeability of connectors based on a change of interface version, vendor, or protocol. Interface version compatibility will be provided through interface adapters and interface inheritance.

Connector Services provides protocol independence
<ul style="list-style-type: none"> • Service interfaces are defined in Java or translated from legacy interfaces • Connector generation tools make it easy to use connectors

CONTAINER SERVICES

Container Services provide processing resources as a network service. Every Openwings platform contains a Container Service that controls access to the platform’s processing resources. The Container Service provides

control of lifecycle management: starting / stopping component execution, and restarting failed components. Other lifecycle features include starting components on system boot, live component updates, persisting component state, and moving components between containers.

Container Services increase system availability
<ul style="list-style-type: none"> • Component Lifecycle • Fail-over • Live updates • Mobile Agents • Clustering

Component restarts or updates can be hot, warm, or cold. Hot updates are accomplished by starting the new component and switching all resources associated with the old component to the new component without interruption. Warm updates are accomplished by notifying the old component’s resources to associate with the new component that has started. Cold updates are accomplished by stopping the old component, then starting a new one.

Component mobility is the key enabler for balancing the processing load over a cluster of Container Services on different machines. It also facilitates agent frameworks where a component can initiate a move to a different platform. A component is moved by pausing its execution, capturing its state, transmitting the state to another container, and restarting it. Capturing the state of a component can be a complex task, involving internal state such as threads, or external state such as services being used and open files. Connections to other components must be terminated and renegotiated; files must be closed and reopened.

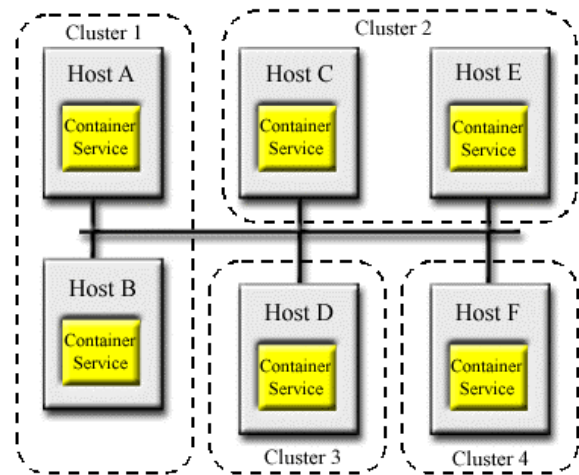


Figure 8. Container Clustering

Clustering of Container Services facilitates load balancing and fault recovery. Container Services in a cluster cooperate to optimize the loading and performance of each, moving components as needed or even starting new component instances to ease the load on a heavily used component. If a

Container Service should fail due to internal error, a system crash, or loss of network connectivity, the other services in its cluster can restart the components it that were running inside of it.

An additional feature of the Container Service is that it provides the ability to add or remove processing resources from a task or set of tasks. Finally, the Container Service is a component that provides system status information, such as CPU loading and free memory. This is the same information used to perform load balancing in a cluster of Container Services.

INSTALL SERVICES

The Openwings Install Services invert the normal installation paradigm. Typical install programs are bundled with the program being installed. As a result, these installers have to ask a series of questions about the platform they are installing the application on. This also poses a security risk in that unauthenticated software is allowed to run on the system unchecked.

The Openwings Install Service runs on the system all the time. The Install Service is aware of settings on the platform and can detect installation files on new media inserted on the platform. In this approach, the user does not need to be queried for platform details and the install jar signature can be validated before anything from the install media is ever run on the platform.

The Openwings Install Service can also resolve context-related parameters through a process called policy resolution. Policy resolution allows deployment environment information to be customized for the component. The Install Service can act as an application server to make components available to other platforms. The Install Service can also handle web-based installation and running programs directly from media. The Install Service allows the registration of listeners for installation events. This allows Container Services to instantly run applications after they are installed, if desired. Figure 9 shows the various states a component goes through during the installation process.

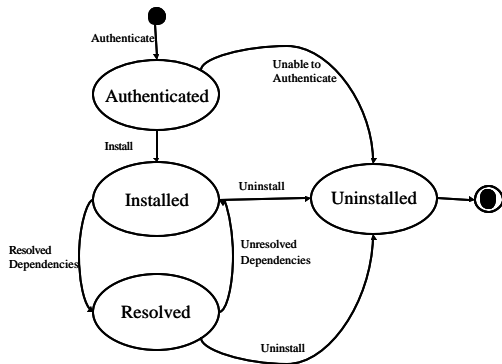


Figure 9. Component Install State Diagram

When given a new installation bundle, the Install Service first authenticates the bundle. It then extracts the contents of the bundle and places it in the appropriate locations on the platform. Next the Install Service tries to resolve dependencies and policies for the component. Once the component is in the resolved state it can be executed. If the user chooses to uninstall a component, the component moves into the uninstalled state and its files are removed from the system. The Install Service provides a voting mechanism whereby interested components can provide input into when a component is uninstalled. For example, a component can register a voter callback with the Install Service that can delay the un-installation of a component until it is stopped.

CONTEXT SERVICES

The concept of a deployment context is central to Openwings. Removing environmental details from components enables them to be easily deployable in any environment.

Traditionally, software has been designed for fixed contexts. When the first computers came out they had custom instruction sets and were isolated from the world. Software had a very controlled and fixed environment to run in. As common platforms began to arise, software had to begin to coexist with other applications on a single platform. The deployment context continued to grow as computers were connected to networks. With the proliferation of compute engines and wireless networking, software is now being deployed in contexts that are increasingly larger and more ad hoc.

Context Services provide an environment for Service-Oriented programming. The following table describes the various operations on a context:

Context Operations	Description
Create Context	Creates a new context, the default administrator is the context creator.
Destroy Context	Destroys a context.
Configure Context	Configures the various behaviors of the context such as discovery mechanism, security, etc.
Create Relationship	Creates a trust relationship between contexts.
Destroy Relationship	Destroys a trust relationship between contexts.

Figure 10. Context Operations

When a context has not yet been created, components reside in something called the default context. The default context provides a Service-Oriented environment for an isolated platform. Every Openwings-enabled platform will have a default context that contains the following services:

Service	Description
Code Delivery Mechanism	Typically this is a http server.
Discovery Services	There can be multiple discovery mechanisms. For peer-to-peer discovery there may be nothing.
Container Services	Provides an execution environment for components.
Install Services	Provides an automated way to install and serve components.

Figure 11. Default Context Services

At the core of a Context is a cluster. Certain services are required to run somewhere in the context, not necessarily on any specific platform. This cluster is responsible for keeping core context services running. These shared network services include:

Service	Description
Network Formation Services	Services necessary to dynamically form networks, such as DHCP and DNS.
Discovery Services	Services necessary for service discovery across the context.
Security Services	Services necessary for authentication and authorization.
Context Service	Contains the state for the context and provides for configuration and control.

Figure 12. Core Context Services (clustered)

The Context Services interact with the Install Service to resolve policies of installed components. Context Services also configure behaviors of most of the other services. All of the contextual information needed about the environment is provided by Context Services.

Contexts allows for components to self-form into systems. For example, a user could establish a context for their entire house. The grouping of components into this context allows them to be managed together, so the user could set all Container Services in the context to act as a cluster. When new platforms are added into the home context, their processing power is added to the overall household processing power. Hence, the overall performance will scale with the number of Container Services installed in the context. In addition, systems-of-systems can be formed by simply establishing relationships between contexts.

Openwings contexts also provide availability features that allow systems to be self-healing. The architecture accounts for the inherent unreliability of networks and services. Leases are used to cleanup services that crash or are lost due to network problems. By allowing services to be clustered across containers and the ability to fail-over to equivalent distributed services, Openwings systems become highly available.

Policies - Key to Zero- Administration Systems

- Policies contain configuration state and configuration rules.
- Policies are resolved to provide a component with an effective policy based on its deployment environment.
- Unmanaged policies can be configured at deployment time.
- Managed policies can be configured at deployment and runtime.

MANAGEMENT SERVICES

Policies provide the key abstraction for zero-administration systems. They allow components to adapt to their environment dynamically without changing core business logic. Policies can be described as dynamic, object-oriented configuration files. These methods allow complex rules for automating configurable behavior. A policy is stored in an Extensible Markup Language (XML) file.

Policies that can only be configured at deployment time are called unmanaged policies. Policies that are configurable at runtime are called managed policies. Policies are used to manage anything from security, to discovery rules, to service lookup rules, and so on.

To achieve zero-administration, policy based management is essential. Openwings provides a simple management model that allows management bean to be published. Each Openwings component contains an extensible management framework that allows management plug-ins to be added and remotely controlled as seen in the next figure.

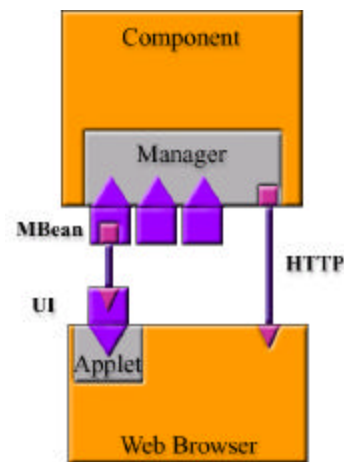


Figure 13. Management Architecture

The framework even allows plug-ins for other management frameworks, such as the Java Management Extensions (JMX) [23] model. This allows these management beans to be published in other management systems. In Openwings, every component, container, and context can be managed.

Control of policies, logging, and components is provided through management beans. These beans are typically used for deployment, debugging, and maintenance. Each management bean can have one or more user interfaces, such as Java programs, web pages, SNMP views, and so on.

Openwings components may be managed individually, or in groups using contexts. In this way, parts of management can be encapsulated at various levels to implement a powerful, flexible management scheme.

SECURITY SERVICES

The growth of the Internet has made consumers aware of the need for increased security. Protection of valuable information assets and digitally controlled physical assets is a critical issue for everyone. The job of securing services, platforms, and networks from malicious users and software is a difficult one. The Openwings security architecture focuses on transport, service and mobile code security. Platform (logins) and network (firewalls) security solutions are available and adequate today. The following features are essential to any security model:

- Authentication – Verifying the identity of a person or piece of code.
- Authorization – Verifying that a user or implementation has permission for the requested operation.
- Data confidentiality – Allowing communications to be private between parties.
- Data integrity – Verifying that the data that is sent does not get corrupted.
- Non-Repudiation – Non-repudiation is the assurance that a principal sending data cannot deny be the sender after sending it.
- Auditing – Detecting breaches in security.

Security Services rely heavily on the Java Security Model, Java Authentication and Authorization Service (JAAS) [25], Public Key Infrastructure (PKI), Secure Socket Layer (SSL), and Certificates. The Java security model provides authentication and authorization of Java implementations through digitally signed code.

JAAS provides for authentication and authorization of users. The JAAS model allows for authentication plug-ins, making it possible to use passwords, smart cards, or biometrics. The plug-in model also allows the Openwings architecture to support both military-grade and commercial-grade security. In addition to the services provided by JAAS, Openwings will implement Role Based Access Control (RBAC). This allows access control lists to be associated with roles and users to be associated with roles. Role based security is fundamental to military security, where the person performing a given role can change frequently.

There are two key services in the Openwings Security architecture: the Authentication Service and the Authorization Service. The Authentication Service handles validation of certificates. The Authorization Service allows the user to dynamically manage access control to the system by simply defining roles, assigning permissions, and establishing relationships with other authentication services.

Openwings enforces three types of security: code security, transport security, and service security. Code security is first enforced by the installation service, which validates installation bundles. Next the container services control the access the newly installed component has to the local platform. Transport security between services is enforced using secure connectors. Secure connectors provide confidentiality and integrity. Connector Services and Component Services jointly enforce service level security. This allows access to distributed services to be controlled.

CONCLUSION

Openwings™ is an open community, non-proprietary effort to define specifications for self-forming, self-healing systems. Motorola and Sun Microsystems established the Openwings™ consortium in June of 1999. Since then, over 100 companies have registered to help mature its development, and more companies are registering daily.

The core Openwings™ framework is designed to incorporate existing commercial standards and is intended for use in both commercial and military environments. Openwings™ is being developed using a community development process modeled after the very successful Java Community Process. Anyone may join the Openwings™ community, participate on expert teams, or use the resulting specifications free of charge by merely signing up at the community web site <http://www.openwings.org>.

The Openwings™ Architecture provides a framework for building plug-and-play, service-oriented, network-centric, self-forming, self-healing systems that are independent of middleware, databases, platforms, and deployment contexts. Openwings™ has a special focus on issues of availability, security, and interoperability.

Openwings™ is the embodiment of a new movement in the software engineering community towards a paradigm known as Service-Oriented Programming (SOP). For an introduction to SOP, refer to <http://www.openwings.org/download/specs/ServiceOrientedIntroduction.pdf>.

REFERENCES

- [1] “Next Generation Networks and DOD C4ISR”: Army Signal Command, Dr. Michael Gentry, Jan. 2000
- [2] IEEE 802.11: <http://standards.ieee.org/catalog/IEEE802.11.html>
- [3] Bluetooth: <http://www.bluetooth.com>
- [4] Cooltown: <http://cooltown.hp.com/papers/WebPresence.htm>
- [5] Mobile IP: <http://www.ietf.org/html.charters/mobileip-charter.html>
- [6] IPv6: <http://playground.sun.com/pub/ipng/html/ipng-main.html>
- [7] Piano: <http://sstg.geg.mot.com/projects/piano/>
- [8] “OpenwingS Visioneering Document”, unreleased
- [9] SEI: <http://www.sei.cmu.edu/>,
- [10] Jini: <http://java.sun.com/products/jini/>
- [11] Chai HP: <http://www.chai.hp.com/>
- [12] Universal Plug’n’Play: <http://www.upnp.org/>
- [13] CORBA: <http://www.omg.org/>
- [14] Ninja: <http://ninja.cs.berkeley.edu/>
- [15] ServiceUI: <http://www.artima.com/jini/serviceui/index.html>
- [16] JSP: <<http://java.sun.com/products/jsp/>>
- [17] JMS: <<http://java.sun.com/products/jms/>>
- [18] CMS: <http://www.omg.org/>
- [19] ACME ADL: <http://www.cs.cmu.edu/~acme/>
- [20] “A Note on Distributed Computing”: <http://www.sun.com/research/techrep/1994/abstract-29.html>
- [23] JMX: <http://java.sun.com/products/JavaManagement/>
- [24] 5 Nines Availability: <http://5nines.mot.com/>
- [25] JAAS: <http://java.sun.com/products/jaas/>
- [26] JCP: <http://java.sun.com/aboutJava/communityprocess/>
- [27] JXTA: <http://www.jxta.org>