Embedlets

Outpost/Embedlets

Architecture Discussion Document

Author: Andrzej Taramina
Chaeron Corporation

Version: 1.7

Date: 3/08/03

Revision History

Date	Version	Description	Author
Jan 30, 2003	1.0 Draft	Initial Draft	Andrzej Taramina/Chaeron Corp
Feb 2, 2003	1.1	First Public release Added general comments on Services Made Persistence Service an Optional Service Added paragraph to Acknowledgements Added design ideas to Event Manager Service Added reconfig comment to Management Service	Andrzej Taramina/Chaeron Corp
Feb 3, 2003	1.2	First post to CVS repository Fixed some typos and grammatical errors Added Dynamic Context/Config to Optional Services Added Revision History page	Andrzej Taramina/Chaeron Corp
Feb 9, 2003	1.3	Added optional Security Service Added Appendix A: Build vs. Deployment vs. Management Processes Added Appendix B: Outpost Management Service and JMX	Andrzej Taramina/Chaeron Corp
Feb 18, 2003	1.4	Merged Config and Context Services into a single Context Service Added note to Lifecycle service that it will co-operate with the Context Service. Added a paragraph to the Event Management Service regarding event queue overflows Added Appendix C: Context, Lifecycle & Persistence Services design discussions Added Appendix D: Embedlets, Services & Adapters Added Appendix E: Lifecycle Dependency Management	Andrzej Taramina/Chaeron Corp
Feb 20, 2003	1.5	Changed Appendix C: Context, Lifecycle & Persistence, to clarify the acceptable use of instance variables & added Servlet vs. Embedlet requirements comparison. Added comparison of lifecycle processes from Component and Container perspectives to Appendix E: Lifecycle Dependency Mgmt	Andrzej Taramina/Chaeron Corp
Feb 28, 2003	1.6	Logging Service elevated to a Core Service Scheduler Service comments updated	Andrzej Taramina/Chaeron Corp
March 8, 2003	1.7	Added contributor recognition to Acknowledgements Added Embedlets logo to cover page	Andrzej Taramina/Chaeron Corp

Table of Contents

Acknowledgements

This architecture document is based on discussions, ideas and input from many contributors, and would not have been possible without such. Key architectural/design contributors include (in alphabetical order):

James Caska
Ted Kosan
Brill Pappin
Chris Smith
Andrzej Taramina
Gregg Wonderly

Much of the Outpost Design Principles section was unashamedly borrowed/adapted from the “Developing with Avalon” document written by Berin Loritsch.

The architecture of the Outpost container and Embedlets owes a lot to work that has been done in the area of Container-based architectures. “Embedlets/Outpost stand on the shoulders of giants”, and were influenced greatly by Servlet Container design (eg. Apache Jakarta Tomcat), EJB Container Design (eg. JBoss), The Apache Avalon project, the Java Management Extensions (JMX), Event and Message-oriented systems (eg. JMS) and general best practices defined for J2EE Container design, amongst others.

Initially it looked like we could use some of the Avalon interfaces/code or base the Outpost Container on a JMX foundation (Mbean Server like JBoss), but those approaches would bloat the container code base and complicate the interfaces. Given the constraints of embedded devices, it was deemed better to use the concepts only, and not the implementations, or to provide an optional “decorator” layer (for JMX Management) to keep the footprint as tiny as possible, and to also ensure that the Embedlet service contracts/interfaces were as simple as possible to make it easier for a wider base of users/developers to learn how to create Embedlets.

Introduction

The purpose of this document is to provide an initial starting point for the discussion of the Outpost Container architecture, the components (Embedlets) that are hosted inside the container and the contracts provided by the architecture (interfaces, services, etc.). It is also intended to provide guidance on appropriate implementation approaches/strategies for various portions of the Outpost Container (eg. Services).

The reader is presumed to be familiar with the following documents:

Outpost Foundation Document – Ted Kosan
Outpost Embedlet Container – Ted Kosan
Outpost Strategy & Marketing – Tech Kosan

and with the discussions that have been taking place on the embedlets-dev list. These documents can be located on the Outpost Embedlets web site at: http://sourceforge.net/projects/embedlets/

It is expected that this document will change and evolve over time, as the design and architectural concepts are evaluated, refined and adopted, and will eventually be a comprehensive overview of the standard and implemented Outpost/Embedlet architecture.

This document is the "architectural vision" of the author, and is based on discussions, ideas and input from many other contributors (as noted in the Acknowledgements section). As such, is not the "end all and be all" of what the Embedlet/Outpost architecture should look like. It is intended as a starting point (since we need one), and will hopefully result in a final architecture that we can all use for guidance as the various modules, services and tools that make up Outpost are defined, documented, developed and implemented.

Outpost Design Principles

Outpost is built on specific design principles. The two most important patterns are *Inversion of Control* and *Separation of Concerns*. Component Oriented Programming, Aspect Oriented Programming, and Service Oriented Programming also influence Outpost. Volumes could be written about each of these programming principles and design mindsets, and the reader is encouraged to do their own detailed research into each design area.

Inversion of Control

Inversion of Control (IOC) is the concept that a Component (Embedlet) is always externally managed. Everything a Component needs in the way of Contexts, Configurations, and Core Services is given to the Component. In fact, every stage in the life of a Component is controlled by the Container that hosts that Component.

Separation of Concerns

The idea that you should view your problem space from different concern areas resulted in the Separation of Concerns (SOC) pattern . An example would be viewing a web server from different viewpoints of the same problem space. A web server must be secure, stable, manageable, configurable, and comply with the HTTP specifications. Each of those attributes is a separate concern area. Some of these concerns are related to other concerns such as security and stability (if a server is not stable it can't be secure).

The Separation of Concerns pattern in turn led to Aspect Oriented Programming (AOP). Researchers discovered that many concerns couldn't be addressed at class or even method granularity. Those concerns are called aspects, are cross-cutting in nature, thus not easily implemented in an OOP inheritance heirarchy. Examples of aspects include managing the lifecycle of objects, logging, handling exceptions and cleaning up resources. With the absence of a stable AOP implementation, the Outpost team has chosen to implement Aspects or concerns by providing small interfaces that a Embedlet Component implements, and through an architecture that supports interceptors at various levels.

Component Oriented Programming

Component Oriented Programming (COP) is the idea of breaking a system down into components, or facilities within a system. Each facility has a work interface and contracts surrounding that interface. This approach allows easy replacement of Component instances without affecting code in other parts of the systems. The major distinction between Object Oriented Programming (OOP) and COP is the level of integration. The complexity of a COP system is more easily managed due to fewer interdependencies among classes, promoting the level of code reuse. One of the chief benefits of COP is the ability to modify portions of your project's code without breaking the entire system. Another benefit is the ability to have multiple implementations of the Component that you can select at build time (or even runtime for those platforms that can support dynamic configuration).

Service Oriented Programming

Service Oriented Architecture (SOA) is the idea of breaking a system down into services provided by the system or container.

Outpost identifies a service as the interface and contracts that are provided to Embedlet components. It is important to realize that a Container is made up of multiple services. To take the example of a Mail server, there are the protocol handling services, the authentication and authorization services, the administration service, and the core mail handling service. Outpost provides a number of low-level services that you can leverage for your own embedded applications. Outpost also provides a number of optional services that can be selected based on the needs of the application and the target platform capabilities. Outpost also provides a "pluggable" Services architecture, so that additional services can be develped and added to the container easily, providing easy extensibility.

Event/Message-Based Architecture

An event (or message) oriented paradigm is a good fit for the Outpost project, since most integration with external devices and external systems (server side) can be implemented with an event-oriented approach. Using an event-based architecture also provides for a very decoupled container environment where the Container, Services, Embedlets, Device Interfaces and Protocol/Transport Adapters (for external system integration/connection) can interact easily through the production and consumption of events, and thus be easily "wired" together using declarative or graphical approaches.

Modular/Extensible Architecture

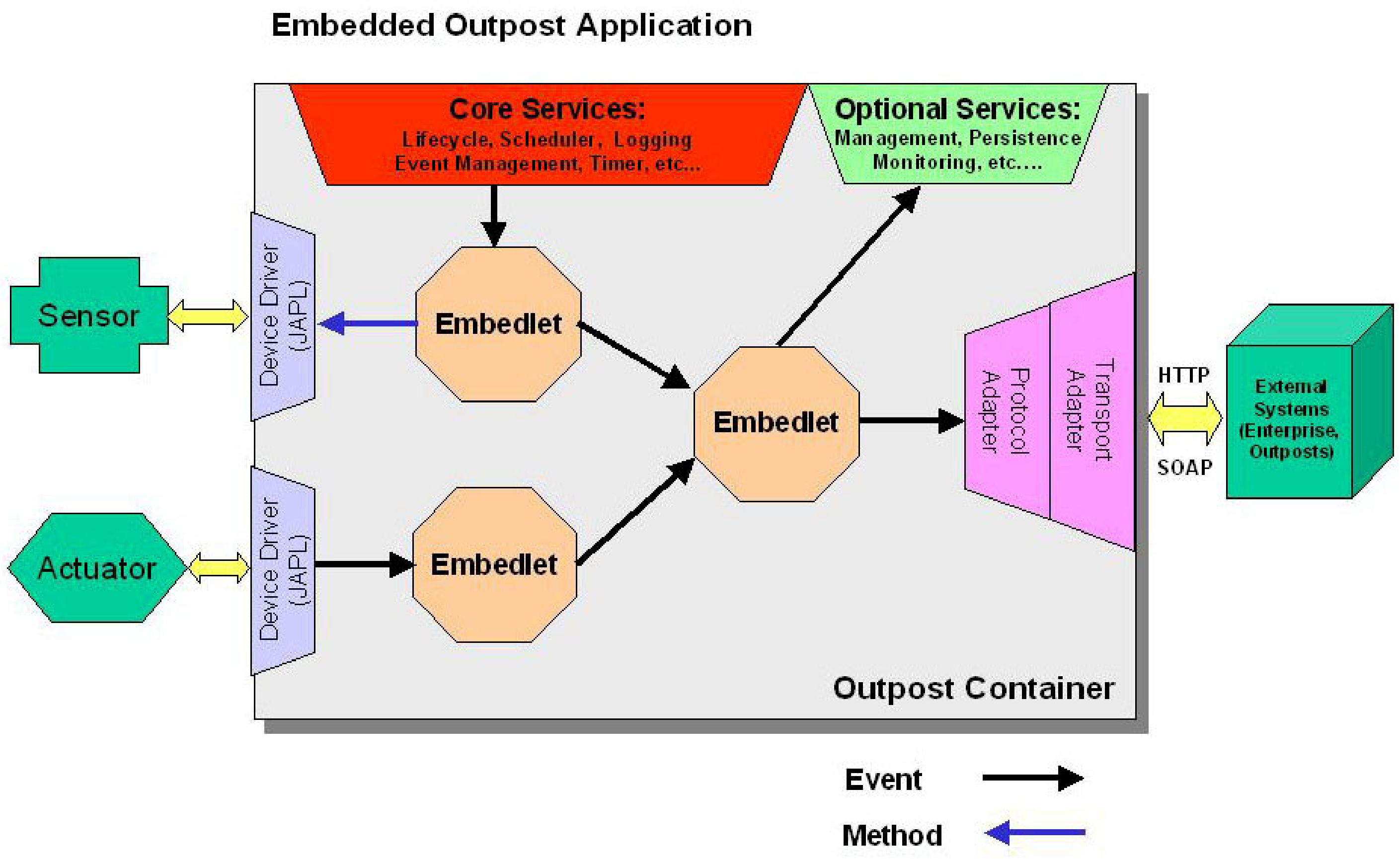
Use of all of the above design concepts results in a very extensible and modular container architecture, which is a key requirement of Outpost due to the constrained nature of the underlying embedded platforms.

Outpost Container

Overview

The services, environment and contracts imposed on Embedlets by the Outpost Container need to be as simple as possible, to take into account the constrained nature of embedded platforms, and to make Outpost development useable and accessible to the largest target market. This means that the services provided by Outpost will be fewer and simpler than those that are provided by enterprise-class containers (such as Tomcat, JBoss, Avalon, etc.), since it will be either not possible or not required to provide many of the services that are needed for enterprise-class containers (eg. Clustering, failover, security, transactionality, dynamic configuration/loading). That being said, the Outpost specification should be flexible enough so that such advanced services could be added in the future as the platforms allow and requirements dictate. This balancing-act between simplicity vs. functionality will be one of the most difficult and critical decision processes that determine the final Outpost archtecture and chances of success for widespread adoption.

An Outpost application will consist of the Outpost Container, Services (Core and Optional), Embedlets, Device Drivers/Interfaces and Communications Adapters (Protocol and Transport). A running Outpost application will look something like the following:



Services

The definition of what an Outpost Service is has been deliberately left a bit vague, to provide increased flexibility and extensibility. Essentially, an Outpost Service is a piece of code that provides common, shared functionality to other Outpost components. Core Services would be an intrinsic part of the Outpost Container, and would comprise those "services" that all Embedlets would typically require. Optional Services could be installed as required, so as to keep the core footprint of the Container to a bare minimum. Embedlets themselves could provide "services" to other Embedlets (where an Embedlet could request a "service" from another Embedlet by publishing a service request event that the Embedlet Service was listening for). This approach provides a very flexible, yet modular and lightweight framework for the extension of and creation of new services that may not be provided as a standard part of the Outpost Container. It follows that Outpost Container Services (Core and Optional) could be implemented as an intrinsic part of the Container (invoked by synchronous method calls or asynchronous events as appropriate) or as pre-packaged Embedlets (which respond to specific events).

Core Services

Core services are those services that must be implemented by an Outpost container and thus are mandatory. It follows that Embedlets can rely on the existence of these core services. These services would typically be invoked through synchronous method calls on the Container itself (which implies that an Embedlet has some way of obtaining a reference to it's enclosing Outpost container), though some core services may benefit from being invoked through the event service (eg. by publishing a standard event). The Outpost container and Core Services should also provide "hooks" at key functional junctions so that custom interceptors could be implemented for unique or extended container capabilities.

Proposed Core services include:

Lifecycle Service – The lifecycle service is responsible for the management of the lifecycle of embedlets (and services). This includes instantiation, intialization, starting, suspending, restarting, stopping and destruction of embedlets. The lifecycle service will also manage dependency relationships, if specified in the XML Config, ensuring that embedlets and services are started in the correct sequence. Embedlets will implement an embedlet interface that will require them to provide the lifecycle methods that the Lifecycle service will call in the correct sequence (eg. init(), start(), stop(), destroy()). The lifecycle service will also provide a way for the embedlet to obtain the context and configuration information during the initialization process (possibly by passing in context/config references to the init() method). The Lifecycle Service will interact with the Context Service so that it can pass the correct contexts to an Embedlet, Service or Adapter at the appropriate points in the lifecycle.

Context Service – This service provides the both the container (Outpost) and Embedlet contexts to the embedlet. These contexts would be passed to the Embedlet (during the appropriate part of the Embedlet Lifecycle) and would provide the Embedlet with information on the environment and context within which it is running, including any configuration information that the embedlet or Service needs. For example, by accessing the Outpost Container context, the Embedlet could discover what optional services were available, what the target deployment platform was and other features such as whether threading was supported or not. By accessing the Embedlet Context, the embedlet could discover special properties that were set in the configuration. These contexts would be primarily defined by XML Configuration files. The Container Context would be accessed through a reference to the container that was passed to the embedlet. Context objects could be created at build/construction time (very possibly code generated into Java classes) to avoid the parsing and memory overhead of having to implement extensive XML parsing logic at runtime.

Scheduler Service – The scheduler service will be responsible for ensuring that each embedlet/service in the container receives it's "fair share" of processor resources. It is possible that there will be two different implementations of the Scheduler Service, one for threaded platforms and another for platforms that do not support threads (co-operative). The Embedlet specification will have to provide guidelines that, in the absence of threads, will enforce co-operative scheduling capability. It is very likely that the Scheduler Service will be highly coupled with or even subsumed by the Event Management Service to provide such scheduling capability, and that the use of an event-driven model for co-operative processing on the part of Embedlets will be necessary.

Event Management Service – The Event Manager Service will provide the ability for Embedlets to produce and consume (register interest) for events. These events could be produced by other embedlets, services (eg. Timer Service) or Outpost Container events. It is very likely that co-operative scheduling of embedlets will be enabled through the use of events, and as such, the Event Management Service will likely be tightly coupled with the Scheduler service (and those two might be integrated into a single service). Embedlets would typically specify what events they are interested in through their XML Config specification, though there may be value in providing a way to dynamically register/deregister interest in specific events at run time as well, for those Embedlets that need more dynamic capability. There may also be value in allowing the target method to receive events to be specified in the XML Config, by event type. The Event Management Service will also define and manage event queues and event priorities. The Event Manager may also need to implement priority elevation based on queue time (measured in events processed), to ensure that all events eventually get serviced, even low priority ones (for example, if an Embedlet is producing high-priority events based on a fine-grained interval timer, it may generate sufficient events at the top of the priority queue so that lower priority events never pop to the top of the queue for processing).

An Embedlet will probably inherit the ability to send/receive events by extending the standard Embedlet superclass. Though event production/consumption is typically an asynchronous process (decoupled through event queues), it may be worth evaluating the need for a synchronous event sending capability, where the sender blocks till the event is processed (eg. synchronous events would not be queued, but processed immediately). By implementing synchronous event processing, almost all Services (whether provided by the Container or another Component) could be invoked using an identical and consistent paradigm, rather than some being invoked through method calls, and others through events. Appropriate caveats and safeguards to the use of synchronous events would have to be put in place, since they could easily compromise co-operative processing on non-threaded platforms (see Scheduler above).

There are situations that could produce the symptom of infinitely growing event queues (if event production exceeds the platform's capability to process events). This should be detectable in most circumstances. One of the drawbacks of event/message based systems is that the event queues are the "buffer" between processes that can run at different speeds, and if one of those processes becomes a bottleneck, all hell can break loose. This is a tough problem to solve. One way to mitigate this issue is by making the Event Manager configurable (eg. max event queue sizes, warning levels and such). When a warning level was reached, non-essential events could be ignored (eg. logging, though what constitutes "essential" or not is a thorny issue. It might be a good idea to specify whether an event is essential or not in the XML config files). When the maximun level was reached, drastic action would have to be taken (eg. throw an exception in response to any attempt to produce a new, non-essential event) until the event queue size dropped below a certain threshold.

Timer Service – The Timer service is responsible for the creation of events based on temporal requirements and should be able to generate timer events for both periodic and one-shot timer values (eg. if an Embedlet needs to poll a device port periodically, it would request a that it receive a specific timer event, and upon receipt of such events would interact with the device). The timer service will be an key component of the Scheduler Service. It needs to be determined if the interface to the Timer Service (ie. requesting timer events) should be event-based (asynchronous) or method invocation-based (synchronous) or both.

Logging Service – The logging service will provide logging facilities to embedlets (and other services). It should probably be based on the Apache Commons Logging API to make it consistent with other common logging implementations. The logging service might in turn send the log entries to System.out (eg. JStamp Charade output, or on a PC JVM platform), or use a Transport/Protocol adapter to send the log entries to an external system. There are a number of ways to implement logging that need to be considered. It could be implemented as an Embedlet, so it could be “wired” in easily with other embedlets that need/want logging services. It needs to be determined if the interface to the LoggingService should be event-based (asynchronous) or method invocation-based (synchronous) or both. A NullLogger implementation will be provided (one that does nothing). Since logging is a Core Service (mandatory), there should be a NullLogger (in the XML Config) if logging was not required for a particular embedded application or embedlet. By doing so, the Embedlet would never has to concern itself if logging is available or not, since it it wasn’t, log entries would just be “eaten” transparently by the NullLogger. Though the Logging Service is a Core Service, there can be many different Logging Services running concurrently in the Embedlet Container, though only one can be designated as the Container’s “default” logging service.

Optional Services

Management Service – The management service would expose management interfaces for the Outpost Container, Services and Embedlets to the outside world using the JMX standard. Such management capability would be exposed through various Protocol/Transport Adapters (most likely HTTP/SOAP initially). Key services (eg. Timer, Event, Monitoring, etc.) and the Outpost Container (which would optionally implement a subset of a JMX Mbean Server) should be designed in such a way as to make it easy to create JMX Management decorators (see GOF Decorator design pattern) that could be "wrapped" around the standard services and container, so that JMX Management capability could be easily layered on top of standard Outpost/Embedlet implementations without requiring any underlying code changes. Though it is possible to have standard Outpost Services implement the JMX interfaces, these interfaces are probably more complex than Outpost Embedlets will require, so a decorator/layered approach makes more sense. The management service would be appropriate for remote configuration, or re-configuration of running Embedlets (though if the optional Management Service was to heavyweight for a particular target platform, a lighter weight dynamic config capability could be created as a "custom" service implemented as an Embedlet).

Security Service – The security service would be charged with providing identification, authentication, authorization and possibly encryption (if the target platform has the resources to support encryption/decryption) facilities. It would typically be used in conjunction with the Management Service to authenticate external management users (very likely using HTTP Basic Authentication, although other userid/password techniques could be deployed). It's an optional service, since deployment of Outpost-enabled processors could take place on secured networks (eg. VLAN on the shop floor) where the network would provide the access control, thus obviating the need for an Outpost-level security capability.

Monitoring Service – Rather than requiring the development of custom Embedlets and code for common operations such as polling of a device port, it would be beneficial to provide a Monitoring Service (implemented as an Embedlet) that could be declaratively configured (using XML Config to specify period, attribute to be monitored, criteria) to periodically monitor a value (device port, embedlet attribute, etc.) and then generate an Embedlet Event when certain criteria were met (value changed, value out of specified range, etc.). This would make graphical "wiring" of Outpost applications much easier by eliminating the need for custom code for such monitoring/polling activities. This service/embedlet would be very similar to the JMX Monitoring Service, and could be wrapped and exposed as a JMX service (see Management Service above).

Dynamic Context/Lifecycle– Certain applications on more capable embedded platforms might need the ability to dynamically specify Context/Config information to both instantiate and configure Embedlets, without having to dynamically load/reload any Embedlet Container and Component classes. This would assume that the necessary classes were already available on the embedded platform. It may be possible to write the Build/Construction-time Config/Context parsing in such a way that the same code base could also be used dynamically at runtime. This is an optional service (or an optional extension of the Lifecycle Service), since dynamic parsing of XML may not be easily supported on the smallest of embedded platforms.

Dynamic Loader – For platforms that provide dynamic loading capability (class filesystem/network storage, dynamic classloading, etc.), it would be useful to have a Dynamic Loader ability so that new Embedlets could be deployed at runtime, rather than at build/construction time. It is expected that Dynamic Loading will not be implemented in the intial wave of Outpost containers, since most embedded platforms will not support dynamic classloading.

Persistence - Many embedded devices will have very little capability (or need) for persistence. However, there are situations where data persistence (eg. multiple configuration sets which could be selected through the Management Service) would be very useful and so an optional Persistence Service is something that should be added at some point. There are many issues attached to persistence, since different platforms have widely different capabilities. For example, some platforms only have Flash Memory available for persistent storage (which survives a power on/off cycle), others might want to send the data to an external system for storage, and retreive the data across a communication link on startup, etc. So any Persistence Service should define an abstract data persistence API that could be mapped to specific platform capabilities, thus decoupling Embedlets that need persistence services from the underlying storage mechanism.

Protocol/Transport Adapters

Outpost will use a “pluggable” approach to communications with back end systems, through the provision of Protocol and Transport adapters. These adapters should likely be implemented event consumers, and would take incoming events, package the event’s “payload” for a specific protocol (Protocol Adapter) and then send the transmission over a specific transport (Transport Adapter). If an event-driven paradigm is used, these Adapters would just be specialized Embedlets that would consume and produce communications-oriented events, making them de-coupled and modular, easing graphical construction. Initially, two Adapters would be included with the Outpost Container implementations, but it is expected that a library of common Adapters would evolve as users created new ones for unique requirements.

HTTP Transport Adapter – Would be responsible for performing communications using the HTTP protocol. There would probably be two flavours of this adapter, depending on whether the embedded application was acting as a client (sending info to a back end) or server (management from outside).

XML/SOAP Protocol Adapter - Would be responsible for taking an event and mapping it into a XML-encoded, SOAP-enveloped document. This adapter would be used with the HTTP Transport Adapter (but could be used with other Transport Adapters such as JMS, SMTP, FTP if they were available) to provide Web Services interface capability to other systems. It would be ideal if the mappings (eg. how to take the data from an Embedlet Event and turn it into XML) were specified declaratively using and XML Config file, since then back end integration could be done declaratively without requiring custom code for the majority of cases.

Device Interfaces

Physical devices (eg. Sensors, Actuators, Ports, etc.) would be interfaced through an abstract interface library (JAPL – Java Abstract Peripheral Library). These interfaces would provide a unified abstraction layer so that all similar physical devices would appear the same to an Embedlet (eg. bit, byte or stream based IO would all use similar interfaces for all devices). This has the benefit of decoupling device-specific code from Embedlet logic, and allowing for declarative (and graphical) wiring of components. The abstract interface library would only expose synchronous, standardized methods, with no concept of polling or timing (polling/timing operations would be handled by an Embedlet or the Monitoring service, thus simplifying the JAPL interfaces). However, JAPL device interfaces could potentially produce Embedlet Events in response to interrupts (if the device/platform supports interrupt driven operations). Configuration of the translation/mapping of a device interrupt to a specific Embedlet Event should be done declaratively if at all possible, to ensure that declarative/graphical “wiring” was facilitated without the need for custom code. The Sourceforge CORK project is actively working on the definition of the JAPL interface definitions.

Miscellaneous

The Outpost Container and Services will require some standard capabilities and these facilities should be implemented using common and avaliable libraries where possible. For example, XML and SOAP parsing/generation might be done using the Enhydra kXML/kXOAP libraries, and HTTP functions could be provided by the kHTTP library. If such libraries are used in by the Outpost Container/Services, then these should be documented and possibly made available to Embedlets that might require similar capabilities, rather than requiring Embedlet developers to use their own selection of libraries, since this would keep the memory footprint much smaller on constrained platforms.

XML will typically NOT be used internally in the Outpost container (eg. for event payloads, etc.) due to the large overhead required for parsing and data storage, which cannot be easily shoehorned onto constrained embedded platforms. XML will be used for external communications, and for build/construction time configuration specification.

Assumptions/Limitations

The initial target operating platform for Outpost is embedded processors which support Java. These include devices such as JStamps, JStiks and PIC processors running the uVM environment, though it is expected that the Outpost container should also run on PC's (for both production deployment and testing). These primary target platforms are characterized by their constraints, and so the Outpost container needs to be as lightweight as possible and not depend on more advanced capabilties (eg. Threading, Dynamic Classloading, File Systems, etc.).

With this in mind, the following assumptions have been made for the Outpost container architecture

Transactions

Transactions will not be initially supported in Outpost (eg. two phase commits, rollbacks, etc.). Any transactional behaviour required is assumed to be implemented in the back end systems that an Outpost container might be integrated with (eg. J2EE or .NET app servers).

Security

It is assumed that Outpost containers will be running on embedded devices that are either standalone or connected to other systems over private networks (LANs, VPNs) for most applications (process control, manufacturing automation, instrument monitoring), and as such, security would provided outside of the Outpost Container or implemented in custom Embedlets by the developer. In this regard, the Outpost container will not initially provide any Security Services.

Dynamic Configuration

Most embedded devices will have very little capability (or need) for dynamic configuration and loading of embedlets (eg. many platforms will not support dynamic classloading), and so the initial release of Outpost will likely not provide any dynamic configuration/loading services. Configuration of the container and embedlets will likely be done at build time. However, the event-driven nature of the Outpost container would allow for the provision of dynamic configuration/loading services for those platforms that support it.

Embedlets

Overview

Embedlet will are small, simple, reusable, modular pieces of code that will be primarily event driven. Like servlets, they cannot run on their own, but have to be hosted in a container that implements the Outpost Container specification and core services. The configuration of individual instances of embedlets will be done using declarative XML-based configuration files, which will allow for graphical "wiring" of embedlets, device interfaces, adapters and a container implementation into a complete, runnable Outpost application.

Embedlets will have to implement a standard Embedlet interface or abstract superclass (yet to be specified) and will have to abide by the contracts, lifecycle and design patterns laid out by the Outpost/Embedlet specification. The contracts and design patterns will be crucial, since many platforms will not support pre-emptive threading (or any threads for that matter), to ensuring that Embedlets will be co-operative in their sharing of processor resources.

Lifecycle

Embedlets will have to implement specific lifecycle callback methods (by virtue of the fact that they have to implement a standard Embedlet interface). The Outpost Container Lifecycle Service will be responsible for instantiating embedlets and moving them through the lifecycle process, calling the embedlets implemented lifycycle methods at the appropriate times. The lifecycle methods that an embedlet will have to implement (some implementations may just be dummy method calls that do nothing) will include:

init() – This method will be called after an embedlet is instantiated, and will allow the embedlet to perform any intialization it needs. The embedlet will be able to obtain a reference to the Outpost Container/Context and it's own Context/Configuration objects to facilitate this initialization. It is expected that the Outpost Container will "register" the embedlet for any events it will consume/produce, so that the embedlet will not have to do this itself. Typically this method will be called during Container startup. The Outpost container will have ensured that all core services are running and available before the init() method is called. All embedlets that this one depends on will have also had their init() methods called, but they are not yet ready to service requests (eg. start() has not been called yet), thus, any embedlets that this one depends on are not guaranteed to be started at this point.

start() – This method will be called when the embedlet is started. An embedlet will not receive any events until after the start method is called. The start() method can also be called after a stop() method has been called, if an embedlet can be suspended and restarted. The Container will ensure that any embedlets that this one depends on will be started before the start() method is called.

stop() – This method will be called when the embedlet is stopped. An embedlet will not receive any events after the stop method is called (effectively deregistered as an event listener). The embedlet can be restarted after a stop() method (if it is restartable), in which case the start method will be called again.

destroy() – This method will be called when the embedlet is destroyed. Typically this method would be called when the container is shutting down. Any resources aquired should be released or cleanup should be performed in this method.

Packaging

Individual embedlets should be packaged in a JAR file, including an XML Context/Configuration file for the generic embedlet (eg. the Embedlet config file could specify what kinds of events it generates/consumes, what optional Container services it needs, etc.). The directory structure and format for this packaging is yet to be described, but should probably be modeled on Servlet WAR packaging standards where possible.

The embedlet should be a standalone package, in that it should not have any dependencies on other embedlets specified within it. Such dependencies and runtime configuration should be specified in a separate and different Outpost Application XML configuration file, to make it easy to instantiate, configure and reuse multiple copies of the embedlet, possibly with a Graphical configuration tool.

Building Outpost Applications

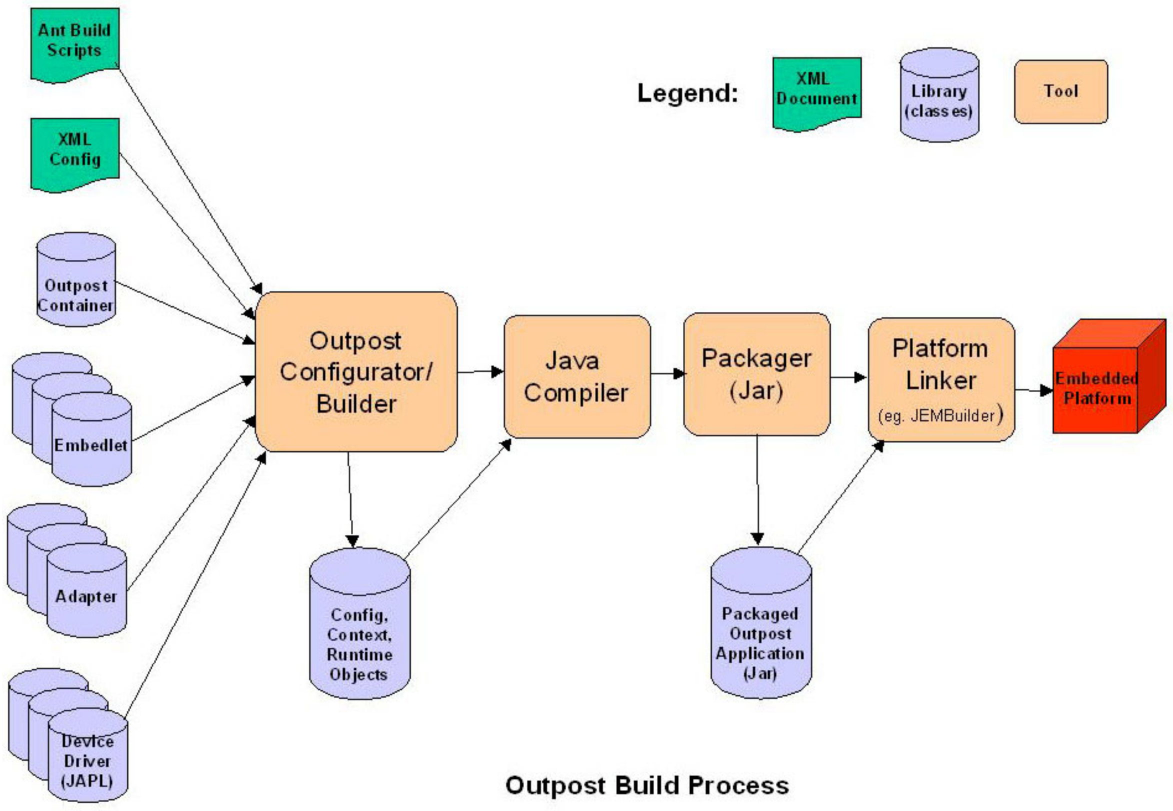
Static Configuration/Construction

The initial wave of target embedded platforms will typically not have the capabilities (eg. dynamic classloaders, dynamic proxies, have enough "horsepower" to do on the fly XML Config file parsing. etc.) to do dynamic loading and configuration, so these tasks will have to be done at build/construction time (typically on a more capable workstation such as PC).

The architecture of Outpost needs to provide highly decoupled components along with declarative specification of the configuration, dependencies, interactions and "glue" that makes up a complete Outpost application implementation, to allow for the build step to "generate" an executable Outpost application. These declarative XML-based config files need to be carefully designed so that they will support graphical wiring capability.

The construction/build process should be automated as much as possible. Ant build scripts and other transformation and code generation tools (eg. XSLT) should take the XML config files, the Outpost Container libraries, individual embedlet/device driver/adapter packages, and link these into an executable application. This linking step would entail code generation of runtime data structures (eg. context, config, container runtime objects) from the XML config files, code generation of "glue logic" if needed (though it would be nice to avoid this), compilation of generated code, packaging the application into a monolithic jar file, creation of platform-specific configuration files (eg. JStamp .ajp project files) and invocation of platform specific linking/loading functions (eg. JStamp JEMBuilder process).

The following diagram depicts such a build process for a JStamp embedded platform:

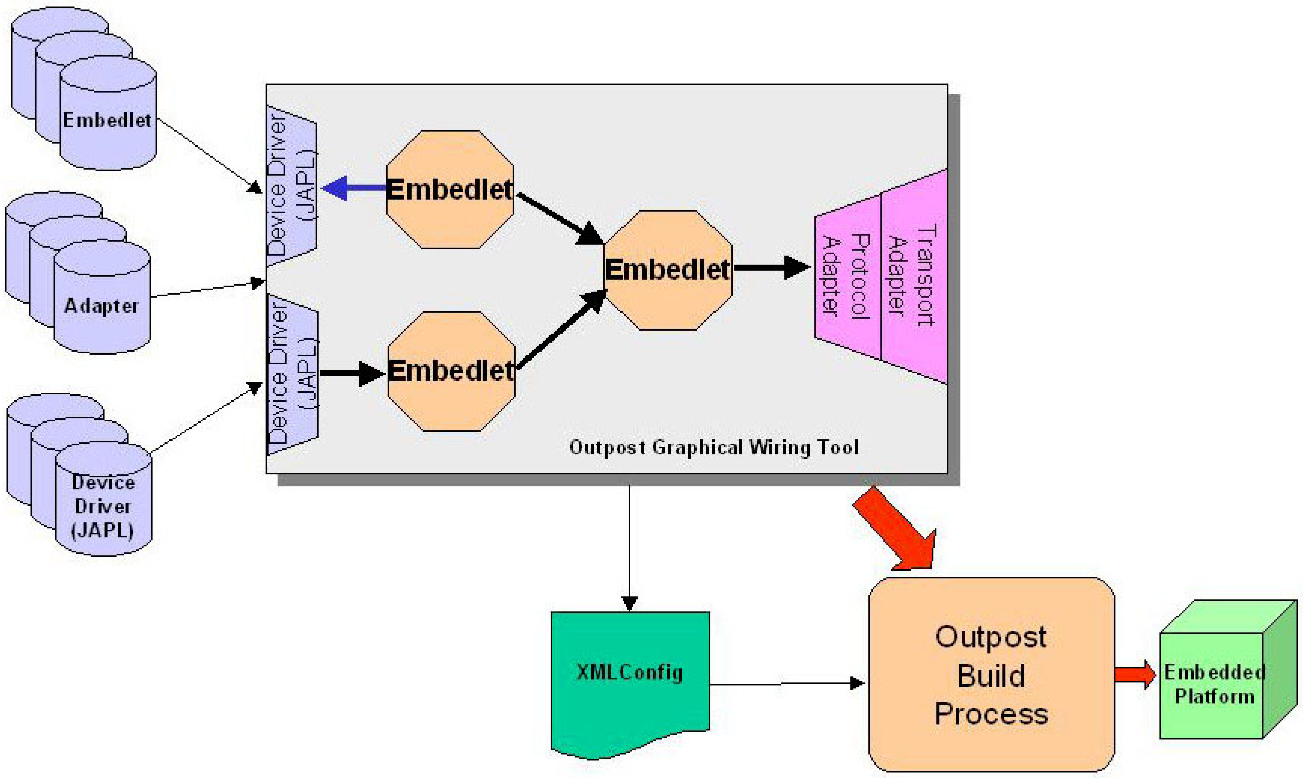


Outpost Build Process

There are many benefits to such a build process, but the primary one is that runtime object (configs, contexts, tables, etc.) are all built and precompiled ahead of time, thus reducing the startup time (no loading, deserializing, etc. of data required) and memory footprint (no need for additional loading/deserialization code).

A tool that can provide graphical “wiring” and configuration capablity is highly desireable. Such a graphical tool would generate the required XML Configuration files, “under the covers”, which specifies a complete Outpost application. Once the wiring was done graphically, and the XML Config files generated by such a tool, the rest of the standard build process would be run by the Graphical Wiring tool on behalf of the user to create the final, runnable application (eg. the Ant build process would be invoked dynanically from the Graphical Wiring tool).

The following diagram shows how the graphical wiring tool would be built on top of the automated build process:



Outpost Graphical Wiring Process

Dynamic Configuration/Construction

Some platforms will be able to support dynamic configuration and construction (using dynamic classloaders, dynamic proxies, have enough “horsepower” to do on the fly XML Config file parsing. etc.), and so the Outpost architecture should not preclude the ability to provide such dynamic configuration/construction capability sometime in the future.

In fact, if properly designed, the Static Configuration/Construction code should be modular enough so that pieces of it’s funcationality could be reused in a Dynamic mode where possible. For example, the parser that reads the XML Config files and creates/generates config objects that are used at runtime, should be packaged in such a way that it could be reused and invoked dynamically if at all possible.

Appendix A: Outpost Build vs. Deployment vs. Management Processes

An Embedlet build process (from the perspective of compiling and assembling an executable), is a different beastie than a Outpost deployment process (which is typically charged with configuration and dissemination of the application), which is again a bit different than the Outpost management process (thinking from an Operations/Monitoring perspective). There is some overlap that forms a grey area (configuration, reconfiguration) that any of these processes can subsume, which seems to be causing some of the confusion on this subject.

The build process will probably be very nearly identical (with variations for specific platforms possibly, JStamp vs TStik vs. uVM, etc) regardless of the application.

The management process will probably be rather unique to each particular application, but by supporting an emerging standard like JMX, existing systems management tools could easily be used to perform this function. The uniqueness here is not found so much in the management tool (use Tivoli, OpenView or even a custom servlet like Greggs, but in all cases JMX provides a base level of commonality on how you technically do the management), as much as unique business procedures (when this alarm is raised, do this.....) which may or may not be automated, depending on the circumstances.

The deployment process will probably be very unique to the particular application that was created. We have seen examples of a custom deployment solution using servlets for particular applications. There probably isn't enough deployment commonality across the whole space of possible applications that Outpost/Embedlets are targeted at to be able to create a single universal "deployer" solution that will meet everyones needs. For example, whether an embedlet/applications needs static versus dynamic configuration capability might alter the thinking of which process should do the configuration, as follows:

If the application is using dynamic configuration of the deployed Embedlets (which is proposed to be an optional Outpost service, described earlier in this document), then the build and deployment processes are quite detatched from each other. You would do one build (per platform), then use the deployment process multiple times to load/config/run the code on many devices (note..the overall deployment process to multiple devices could be automated, basically in a loop that runs through the single device deployment process for each device). The dynamic configuration would be performed using the optional Outpost Management Service...which could be easily implemented as a Browser-visible Servlet that issues (under the covers) the JMX management calls to do the config for each device. The architecture handles it easily, and the build and deploy are nicely separated in responsbility.

If we look at more constrained devices or applications that use static configuration (that may not be able or want to support dynamic configuration) , there would probably be a bit more interaction between the build and deploy processes. The build process, which might include generation of the static config, could be used as a "subroutine" of the wider scoped deploy process. For example, the deploy process might grab the config info from a database, then call the build process (which could be just an JVM process which runs Ant) with some parameters specific to the device being deployed/configed, and the build process (Ant) would create a custom static build based on those parms. The deploy process would then grab this output (the statically configured build) and would somehow install it on the device. So even in the static scenario, it's possible to decouple the very application-specific deploy process from the build, by treating the build as a self-contained "subroutine".

Ant is very powerful for build processes, especially Java and XML-oriented ones (it comes with a large library of tasks to handle very common Java and XML activities, including XSLT transformations, which might be useful for any static code generation we need).

It's yet to be determined whether Ant also provides enough compelling value to a "deployment process" (that will be unique to each user's application and environment) to make it worthy of consideration. Out guess is that Ant will not be as useful for that, and that more custom, UI-based deployment solutions will be required for many scenarios.

That being said, there may be some situations where there would be benefit to having the Deployment tool generate a custom Ant build.xml script for each device that needs to be deployed, based on a template script. This might provide for a more flexible custom deployer solution, as compared to one that hard-codes the deployment process in Java (or some other) code. Conceptually, this hybrid approach uses Ant as a rudimentary BPM (Business Process Management) engine with the businesss process (deployment process in this case) being declaritively specified in XML files (templates, etc.). For large Fortune 1000 companies with massive deployment requirements, it might even make sense to use a commercial BPM for this task, rather than investing in the development of a custom solution.

More philosophically speaking, regarding the design of business systems, declarative specification of processes is much more flexible and maintainable, at lower cost than procedural (eg. programmed/coded) implementations, for complex processes. Also the design practices/paradigms of Separation of Concern, Modularity, Layering and Encapsulation apply to the build/deploy/management process implementations as much as to the Container design itself. In a nutshell: "keep 'em all decoupled as much as possible".

Appendix B: Outpost Management Service and JMX

If you compare the JMX standard with the Embedlets Architecture (as outlined in this discussion document), you'll find there is a lot of conceptual similarity (deliberate on our part ;-)) between Embedlets and JMX. At a conceptual level there are a lot of parallels between the two architectures, with the mapping (conceptual, not intended to imply inheritance or implementation dependency) looking something like:

Outpost/Embedlets	JMX
Outpost Container	MBean Server (and managed resource itself)
Outpost Services	JMX Agents/Services
Embedlet	MBean (managed resource/properties)
Event Manager Service	JMX Notifications
Timer Service	JMX Timer Agent/Service
Monitoring Service	JMX Monitoring Agent/Service
Protocol/Transport Adapter	JMX Connector/Adapter

Looking at the JMX spec, external management is done through a protocol/transport adapter, which exposes the JMX interfaces/API's at the management end. This means Outpost can implement some hooks in the Outpost Container and Services, along with a custom Protocol/Transport Adapter (probably using Web Services, eg. SOAP/XML/HTTP) that exposes the Embedlet management functions outside the device. Note that we do NOT need to implement the actual JMX interfaces in the embedded container! On the management side (management console, server side JMX-aware management tool like Tivoli for instance) all Outpost needs to do is provide an adapter that can translate the Management Protocol (XML vocabulary) into the real JMX interfaces, classes and methods. This can be done because this architecture deliberately maps nicely onto JMX concepts as shown in the above table.

So Outpost provides the best of both worlds, optional, but very lightweight Management Service on the embedded device, with an external JMX-compatible (but decoupled) adapter and tools (like a Process Control UI remote app) layer that translates from the Outpost Management protocol to JMX standard APIs (and vice versa).

One of the benefits of this, is anyone writing external control or management applications can use a standard API to do this, namely JMX, which is gaining ground very rapidly. If there is a standard way of doing something there is a lot of value to adopting that approach, since it widens the market and gives Outpost more credibility/acceptance value. Using this approach, the glass-house server operations staff could actually use Tivoli, OpenView, Unicenter, etc. to manage the distributed Embedlets. In fact, this approach would let operations staff (which could be the shop floor operator, foreman, plant manager, etc.) receive real time alerts if things start to go amiss, and receive those through the corporation's normal management infrastructure. This would be a sales/adoption feature, big time! Management is/will be a big issue with thousands of Embedlets flung around a corporation and integrated to the back ends.

With the solution that is envisioned (see above), with the Management capability exposed as a Web Service (SOAP/XML), it would be trivial to integrate into any other back end technology that was not JMX enabled, yet still provide tight integration with JMX-based management solutions where desired, with minimal performance or footprint penalty.

A JMX/Embedlet Adapter would defintely be a huge boost to the acceptablility at the Enterprise management level. Even if such a feature is not deployed right away (or ever) it's a very, very big selling feature. Remote management of Embedlets, to the point of enabling integration with the large, common back end management systems (eg, Tivoli) would be a HUGE benefit for Embedlets. We would even say it would be more important than many other features of Outpost. Manageability is a key issue for corporate systems when they become integrated together (which is one of Outpost's key goals).

JMX MBeans would not necessarily have to be implemented/running on the embedded device. With a Web Services/XML/SOAP management “on-the-wire” protocol that provides 'events', 'properties' and 'actions', Outpost can write a single MBean that can run on the managers device, and talk via a serial network to the device. This would facilitate remote management in places where device have a network to somewhere else, and a serial port could be used for management of individual devices.

More capable platforms (maybe JStik's or PC's) might want to actually implement MBeans on the controller (there are some benefits to that approach, and it could be implemented as a decorator layer without touching the core embedlet container code), but for the most part, a decent XML-based management vocabulary, exposed as a web service over HTTP will give Outpost more flexibility, even to the point of providing management across serial cable connections (or wireless equivalents like Bluetooth). The possbilities are very exciting and would make a very compelling "sales pitch".

There is value in using a SOAP-based RPC approach to the XML managment schema/vocabulary since it maps nicely to the remote RPC-like JMX API's cleanly and there are a lot of back end tools that let you create/consume such Web Services RPC requests without having to bother with the SOAP/XML "plumbing".

However, the user wouldn't/shouldn‘t have to know anything about the underlying XML syntax, since this would be automatically assembled and then parsed at both ends (JMX API's used by the remote GUI control app to issue the request through the JMX Adapter, then across the comm link and the to the Outpost Managment Service API used by the Embedlet to respond to the request).

Such genericity has to be implicit in the design of the Outpost protocols and XML vocabularies/schemas for the container to be able to provide useful, generic, reuseable services.

The Outpost Management service would also use the optional Security Service to provide identification and authentication so that only authorised personnel were able to perform management functions, thus changing the operations of an Outpost Container running on an embedded controller.

Management is something that should be layered transparently on top of an Embedlet (and the Container and it's services), so it can be pluggable. What is exposed would be specified declaratively in the XML Config files, NOT in code (though we might consider exposing the Management service through API calls as well for those that are writing plugin services). Methods (be they property getter/setters or any arbitrary method an Embedlet implements) or events (translated into external “notifications” in JMX parlance) would be tagged as "Manageable" in the outpost.xml config file and thus made externally visible without requiring any code on the part of the Embedlet developer/deployer. The "hooks" or “glue” to let the Management Service invoke these methods would be created during the build/deploy cycle, not in handcoded embedlet code.

Appendix C: Context, Lifecycle and Persistence Services

There are some fundamental differences between Context, Lifecycle and Persistence services, even though they seem related, and may all be needed to address certain application requirements. That being said, the intention is that these should be independent and decoupled services that can be used standalone, or in combination with other services. This approach is also in keeping with the design goals of keeping Services very modular, simple, and focused on core functionality with little overlap between services. It is better to have many simple services that can be composed into more complex services, as required, than to create one or two monolithic services that do everything under the sun.

This Appendix discusses some design thoughts and criteria as they relate to these particular services.

Context Objects Versus Embedlets

Many attributes (especially initial config info) of Embedlets look very much like JavaBeans. And that is exactly how they will be implemented...but using a generic property capability (rather than method naming standards and introspection as JavaBeans do, meaning we will have only two methods (though the spec may want to restrict property values to strings only):

```
void setProperty( String name, Object value );

Object getProperty( String name );
```

That's what the context (eg. config) objects that are given to Embedlets will do. They will simply be data repositories....with NO business logic or process, and so they are easily serializable without impact on anything else. The embedlet code would then provide the business/process logic, including interfaces that the lifecycle service would invoke.

It is worth noting that the context objects (which contain all the config info that the container or embedlet needs) should be serialized and not the embedlet itself (meaning instance variables). A context object would not have any references (from a startup perspective) to non-serializeable objects so reconstituting it is much easier...no custom serialization code required. All such context objects can be thawed in one fell swoop, ignoring init/lifecycle requirements, since there wouldn't be any. Then during the initialization process (controlled by the container lifecycle service) the Embedlet is given it's context objects, with which it can initialize things like initial state and/or instance variables, connections, JAPL devices, etc. (and very likely store references to these dynamic resources using the generic properties capability in the context object).

It is worth comparing Embedlets versus Servlets because though they have many similarities, they also have many differences (comparison adapted from comments by Chris Smith, Oopscope):

Servlets requirements driving their design:

1) Small number of potentially lengthy procedures
2) Massive number of simultaneous users
3) No real time requirements
4) Little inter-servlet communication

Embedlets requirements driving their design:

1) A potentially large number (dozens) of compact, granular process components with few instance variables.
2) A limited number of users/clients (the internal threads + external config/query)
3) VERY real time requirements which has to a main design consideration for Embedlets to succeed in the embedded market.
4) Heavily interconnected components.

Embedlets (as are server-side Servlets) are not typically, nor should they be data-based components (which represent real world entities, though Embedlets will need to maintain state information in many cases), unlike JavaBeans which are primarily data components (though they have an ugly event/change propagation system that is rarely used these days). If there is no requirement to persist the intrinsic “state” of an Embedlet, then it is acceptable to use instance variables in the Embedlet class to maintain such information (since this addresses the specific Embedlet requirements noted above). However, Embedlets should not typically use instance variables to store state if that state needs to be persisted (to implement highly-reliable/restartable systems and the like). Persistable state (and other) information should be stored in the Embedlet Context or similar object, using the generic property/value methods shown above, since this makes it easier to serialize and store/retrieve such state information.

In summary, serialization (regardless of encoding style) is great for data components. It sucks for process based components.

The Embedlet specification thus splits these responsibilities into two related but decoupled pieces:

1) Context objects = Data-based components that are easily serializable since they have no temporal/process semantics.

2) Embedlets = Process-based components which are easily managed in a temporal lifecycle since they have no intrinsic data semantics (barring some simple state management) and thus have little need for serialization. If serialization of the Embedlet state is needed then this state information should be stored in the Context object and not in instance variables.

By splitting it up we get the best of both worlds, and we get the ability for the container to manage threads (or not if they are not supported) almost for free.

Context Service and Objects

The Context Service will provide two (or maybe three) different context “layers” that will be a mandatory part of the spec. Embedlet application developers can provide additional layers themselves if they find that is needed.

1) There would be a Container Context object (with properties of course) that the container would use to configure itself, and where embedlets could get global environmental info (eg. what services are available). Embedlets could also insert their own properties into the Container Context (with a suitable naming convention for the property name to avoid collisions) to expose information across the whole application. Some of these properties would/could be set in the outpost.xml config file.

2) There would be an Embedlet Context object for each distinct Embedlet. It would have all of the properties that the embedlet needed to know about or save (the short term properties persistence we have been discussing). Some of these would/could be set in the embedlet.xml config file. If there is no requirement to persist specific properties/state/data items, then it will be acceptable to use instance variables to store this information, rather then keep it in the Embedlet Context.

3) It may also be handy to have the ability to store arbitrary properties attached to an Event. An Event Context if you will. But it’s not certain we really need this level, as there are other ways of passing this type of information.

Persistence Service

When we talk about a Persistence Service, we mean something that has a life outside of the Embedlet lifecycle, and could provide persistent storage (of such values, and maybe any other arbitrary objects) that could survive power recycles, etc. Short-term storage of properties for an Embedlet is a core function provided by the Container's implementation of the Lifecycle and Context Services (as part of the spec and contract/interface the Embedlet implements). The Persistence Service (eg. file system) would be an optional service, since many applications will not require such a service.

The Persistence Service's mandate is to provide long(er) term storage of data. It has nothing to do with temporal or lifecycle processes like initialization (of hardware, threads, etc.), and is not intended for that purpose.

The method and encoding (serialization) that the Persistence Service uses to provide this long term storage/retrieval should be transparent to any Embedlet (or other Service/Adapter) that has need of persistence. That is, the storage/retrieval API would be consistent, but the underlying implementation would be "pluggable". This also implies that a Container should be able to host multiple instances of a Persistence Service. For example, one implementation might store objects to flash, another across a network connection to a back end repository, etc.

There is merit in using the generic property/value approach (mentioned in a prior section) as the preferred method for exposing objects that are persistable, since it does not require introspection/reflection. The KISS principle leads us to believe that the first implementation of the Persistence Service API should recognise objects (like Context objects) that implement such properties (denoted by the implementation a specific Properties interface or base class) as being persistable. This should be sufficient for most Embedlet applications, but should be revisted as real-world implementations are deployed in case a more sophisticated approach is warranted.

Serialization

The concept of using serialization for the creation of Context objects (deserialization) and for object Persistence (serialization for storage, deserialization for retrieval) is a viable technique for implementation of these services.

Serialization can be implemented in many ways, however. There is the classic JavaBeans serialization that relies on method signatures and introspection/reflection (which may not be available on all embedded platforms, eg. aJile processors do not yet support introspection/reflection, and if XML is used as the underlying encoding, the parsing overhead may be too heavy a burden for constrained platforms). Creation of Context/Config objects from XML Config files is a form of deserialization, which might make sense for many scenarios. One powerful technique is to code generate actual Java objects/classes at build time to create Context objects, in effect, doing serialization/deserialization through Java Code and bytecodes. Another encoding worthy of consideration (if the original data is XML-based) is to encode serializated objects pre-parsed as SAX events, since this can leverage the benefits of XML, yet keep the parsing overhead down to a minimum for certain applications.

Though all of these techniques have their place, they should be transparent to the Embedlet application code in any particular Container implementation that conforms to the Embedlet standard. In this way, Container developers can weigh the pros/cons of different serialization approaches without impacting the ability of Embedlets to run in that container.

The point is that serialization is great for packaging static data, but is not good for representing dynamic, temporal processes. What we have tried to do is propose a solution that allows for serialization for what it's good for (config info, which is just static data for the most part) and doesn't try to use a hammer to pound in a screw (using serialization for the temporal lifecycle intialization process). The container specification will allow the use of serialization as a packaging solution for those container implementations that wish to use it.

Doing this allows the flexibility to decouple configuration encoding (code gen, serialization, etc.) from lifecycle intialization, and thus making it a container implementation choice that is transparent to an embedlet.

Example of a composite Dynamic Configuration Embedlet/Service

Though the Config, Lifecycle and Persistence services seem quite simple when looked at separately, they could easily be combined to provide much more complex composite services.

Chris Smith of Oopscope and Gregg Wonderly of Cyte Technologies proposed a scenario where custom configurations might need to be stored, selected and the container/applicationdynamically reconfigured on remote embedded devices. With the help of the Management Service (to provide external control), it would be a fairly simple matter to write a Dynamic Configuration Embedlet (or user-defined service) that could be used to take existing configurations (Context objects) and store them in flash using the Persistence Service (for instance). This Embedlet could also then retrieve the a specific configuration (Persistence Service), which would be deserialized into a Context Object (as defined Context Service), and then would issue Lifecycle Service invocations to stop specific embedlets, and then restart them with the selected configuration information.

Another scenario might entail standard configurations being stored remotely in a repository (eg. DBMS or server-side servlet). An operator could reconfigure the device remotely using the same composite Dynamic Configuration Embedlet, but one that used a network-based Persistence service to transparently obtain the config object from the remote repository.

The use of small, modular, independent Embedlet Services provides a lot of flexibility in creating new and more complex services/embedlets through composition (without increasing the minimum Container footprint for those appliations that do not require such advanced capabilities). It is hoped that a library of such re-usable embedlets/services will spring up around the Embedlets project (similar to the JAPL device driver library project), that developers could download, customize and deploy to create custom services.

Appendix D: Embedlets. Services and Adapters, Oh My!

There is a lot of similarity between Embedlets, Services and Adapters (and Device Drivers such as JAPL, though these will not be discussed here). Under the covers, all three of these constructs could be implemented using shared code and identical facilities (eg. they would all conform to the standard lifecycle process of init(), start(), stop(), destroy()).

This begs the question of why distinguish between these three constructs, since it is not yet certain that we will need such differentiation? There are a number of reasons why such a taxonomy might be beneficial:

1) Some services may be implemented as an intrinsic part of the Container implementation, and thus not be packaged or coded like an Embedlet would be.

2) There is value to the embedded application developer in using such a classification scheme (to the point of implementing differently named interfaces or base classes, even if the functionality/signatures are the same), since this "labelling" clearly denotes the "function" of the component. Services provide re-useable, generic functions that typically would/could be shared by different applications/embedlets. Embedlets provide the "business logic", and Adapters provide connectivity to the outside world. In this way, it promotes "best practices" modular development of applications to some degree.

3) As the Embedlet specification and container implementations are adopted and deployed, we may discover that each of these three components could benefit from the provision of some unique API signatures and functionality (we have a gut feel this will be the case expecially with Protocol/Communications Adapters)

4) It's easier to start with the three categories right from the beginning even if their base classes are virtually identical initially. There is little cost to this (in memory footprint or processing) and it provides future expandability if/when that becomes discovered/required.

5) Graphical wiring tools would intrinsically know how to spatially locate the components on the wiring board. For example, Services would be along the top edge of the window, Device drivers on the left edge, Adapters on the right edge and Embedlets in the middle.

6) Temporal dependency management (as managed by the Lifecycle Service and Container Startup process) is facilitated. All core services would be started first. Then all Optional Services and Adapters would be started next (based on dependencies between themselves, since some Optional Services may depend on Adapters and vice versa) and finally Embedlets would be started. This ensures that the Container is fully functional and all services required by an embedlet are guaranteed to be available once it receives it's start() invocation. This provides a clean distinction between Container initialization/startup and application (eg. Embedlet) startup.

7) This is consistent with current "best practices" in container design.

For these reasons, the first Embedlets Specification (and Container Reference Implementation) will provide such component-level distinctions.

Appendix E: Lifecycle Dependency Management

An Embedlet container will need to provide dependency management as part of the Lifecycle Service. This capability is especially crucial during initialization/startup of the Container and application, since Embedlets may depend on each other and on Services/Adapters to provide various capabilities.

There are a number of dependency scenarios that need to be considered:

1) Components might explicitly depend on other components (eg. An Embedlet might need a certain Optional Service to be installed). A component should not be initialized (init() lifecycle method invocation) until all of it's dependents have been initialized (though at this point the dependents are not "started" an so not usable). A component should not be started (start() lifecycle method invocation) until all of it's potential event consumers have been started (meaning the consumers are ready to respond to events). Explicit dependencies will likely be specified in the Outpost XML config file (which provides the "wiring" definition for the intended application). It will be the Lifecycle Service's responsibility to ensure that components are initialized and started based on such dependencies. This level of dependency will typically be validated (eg. a required service is included) at build time.

2) Since Outpost is an Event Driven system which aims towards declarative specification of application "wiring", various components will either consume or produce events or both. Typically, consumers of a particular event should be started before producers of that event, to ensure that events are not "missed". It is expected that which events an Embedlet consumes/produces will be specified in the outpost.xml config file (application wiring definition). This would allow the Lifecycle service to start up Embedlets in the correct sequence automatically. There is a tricky case where event propagation can loop back (typically when there is at least one embedlet that can both consume and produce events), where the sequence is not determinable automatically. There should be some manual way of specifying startup order (in the config xml file) to allow for such situations. It is intended that the Container will automatically "register" consumers of specific events, based on the XML configuration information, and will call a specific event receive method on the Embedlet. Identification of Producers is a bit more problematic, since the Embedlet typically needs to generate events based on application logic. It might be best if an Embedlet was required to specify which event types it can generate (in the XML config) and at runtime, check that it is only producing those events that were declared (and throw an exception otherwise). This would make the job of the Lifecycle and Graphic Wiring facilities much easier.

3) Dynamic reconfiguring of components (stopping, reconfiguring and restarting) should also take into account any dependencies. For example, a Service should not be stoppable if there are Embedlets running that depend on it. This might be complex to implement at runtime, and so dynamic reconfiguration may be a feature that is not provided till later in the release cycle.

This lifecycle can be viewed from two different perspectives, the Component;s perspective and the Container's perspective.

From an Embedlet's (component's) perspective the lifecycle looks as follows:

1) constructor() - the embedlet constructor is called.
2) initialize() - get ready, configure hardware/JAPL, acquire resources, initialize state/instance variables, etc.
3) registerEventListener – wires components together. **NOTE:** this is transparent to the embedlet, since the container will do it on the components behalf, thus this is not a method invocation on the embedlet.
4) start() - do any startup stuff you need, including managing state or issuing initialization events
5) stop() - shut down event generation
6) terminate() - clean up and release resources held.

Note that the registerEventListener should be done by the Container, on the embedlets behalf, prior to the start(), since if an embedlet posts an event during the start() method, it could end up generating an event (in an already started embedlet) that the original embedlet should receive (this is circular, but might be valid in some circumstances).

However, this lifecycle looks a bit different from the Container's perspective:

- ♦ The container would perform steps 1 & 2 for all components, in dependency sequence. So if your embedlet specified that it was dependent on others then the other components would go through lifecycle steps 1 & 2 before the embedlet went through 1 & 2. This way, an embedlet knows that all it's dependencies have been initialized (but not started) when it's initialize() method is called (and it can thus check for existence, various context parameters, etc. of it's dependencies, if it needs/wants to).

- ♦ Then the container would do steps 3 & 4 in event consumer/producer order. That is, it would start all event consumers before event producers for any particular event. This makes sure that if a producer posts an event in it's start() method (event posting would be prohibited in the initialize() method) it is guaranteed that consumers are already listening.

The Embedlet Specification details what the responsibilities are for the Lifecycle service (and how such dependencies are identified in the xml config files) with regards to each of these dependency scenarios.