# Argos, an Extensible Personal Application Server

Arne Munch-Ellingsen, Dan Peder Eriksen, Anders Andersen

University of Tromsø, Norway

**Abstract.** Argos is a microkernel based, small-scale or personal middleware container that is extendible through deployment of system services. System services to support development of end user applications in sensor network, pervasive, context-aware and mobile setting have been developed and used to easily allow for application development of user application in this domain. Argos also gives enterprise container type support to user-centric application development, without the complexity and limitations enforced by enterprise containers.
Annotations, notifications, reflection, dependency injection and hot deployment are together used to create the Arogs run-time extensible and adaptable personal container.

## 1 Introduction

Traditional application servers support business applications and have a focus on scalability, integration, transaction management, safety and security [1]. Such enterprise application servers are well suited for enterprise applications that need this kind of system support. However, a large group of applications does not fit this model. Their demands are different and possibly highly specialized. One approach to create such applications is to start from scratch and integrate all needed services in each application. Typical such applications are found in embedded systems, in sensor networks, in context aware systems, and in personal or small-scale systems, often with the mobile phone or the laptop computer as the end user terminal.

Argos supports these kind of applications. Some requirements in these settings are similar to requirements that appear when developing enterprise applications. Examples include support for persistence (database), general web support (web server), and support for web service interface. Other requirements are domain specific or linked to the fact that such applications can be user-centric. One such important example is access to local resources (file system, sensors). Another observed aspect is that not all applications need the same system support: "one-size-does-not-fit-all". This observation led to the design and implementation of Argos, an expandable small-scale or personal application container with a minimal microkernel core [1].

[1] We will refer to the "microkernel core" as the "core" in the rest of the text

The minimal Argos core can be extended for different application domains. The development of Argos applications should focus on the actual logic of the application and expect specialized support from the container at run-time. Argos supports rapid development of specialized applications in supported application domains. Argos is an extensible application server. Its minimal core supports life-cycle management and a few other core services. In a given setting this core is extended with system services implementing support needed by its applications. Such system services can provide persistence, bindings (to be able to bind to information sources), web services, and so on. The application programmer develops application components as Java objects (POJO, Plain Old Java Objects) [2]. Such components can specify application server and system component dependencies using annotations [3]. Examples are annotations specifying that a given method of the object should be invoked every 10th second (lifecycle) and that the current value of a given attribute of the object should be stored in a database (persistence).

The provided specialized application support makes it possible to create domain specific container configurations with Argos. Since Argos supports user-centric and small-scale systems, its users often refer to the Argos container as a personal container. In short, Argos is both a personal container and it can be used to create domain specific container configurations.

## 2 Argos core

Argos is a container based middleware system and the Argos core is the minimum configuration that defines the Argos container. The Argos core defines a service and component model, including component lifecycle handling, implicit instrumentation (for monitoring and control) and a set of supported annotations. In the default configuration of Argos, a service is a collection of components, web pages (one possible presentation), desktop widgets (e.g. Yahoo! Widgets, another possible presentation), mobile applications (j2me or mobile cf .net applications), instrument panels (to monitor and control the service), and external wrappers (not discussed in this paper). An Argos component is a POJO class with additional meta information expressed using Argos supported annotations. The annotations are part of the default programming model offered to system services and user application programmers. The set of supported annotations can be increased through deployment of new system services. The Argos core container allows deployment of system services and user applications. System services are used to augment the intrinsic capabilities of the Argos core.

A system service implementation can be replaced with a different implementation providing the same kind of service. This can be used to replace a system service implementation with an improved implementation. It can also be used to replace the service with another implementation better supporting the current application needs and the current environment or setting of the application (adaptation).

User applications depend on the functionality of the Argos core and the deployed system services to create end user applications. The Argos core supports "Hot deployment" of system services and user applications. Argos deployment is described in more detail in section 3.

Figure 1 gives an overview of the Argos core with the default set of system services. It is of course possible to start Argos with no extra system services, but the default set of system services represents an often used configuration. Jetty [4] is an embedded web server, and Hibernate [5] and Derby [6] together provides persistence (Derby is a database and Hibernate is an object/relational persistence and query service). The web service system service provides web method (SOAP[7] and XML-RPC[8]) access to component methods. A more detailed description of the different system services will be given in section 4.

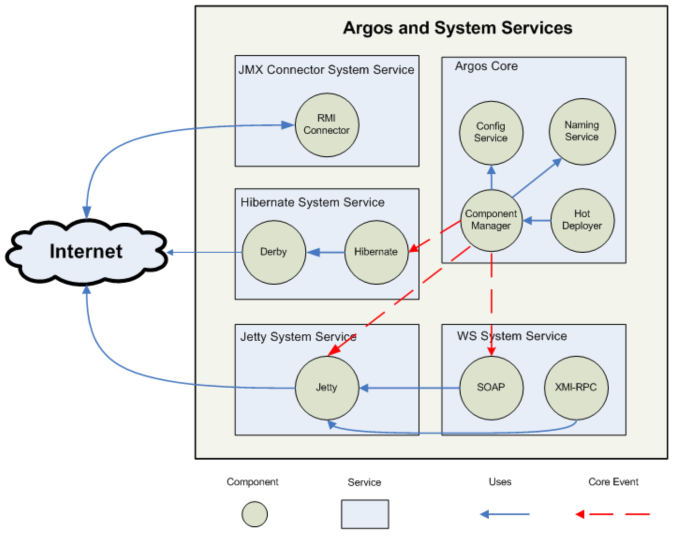


**Fig. 1.** Argos Core overview

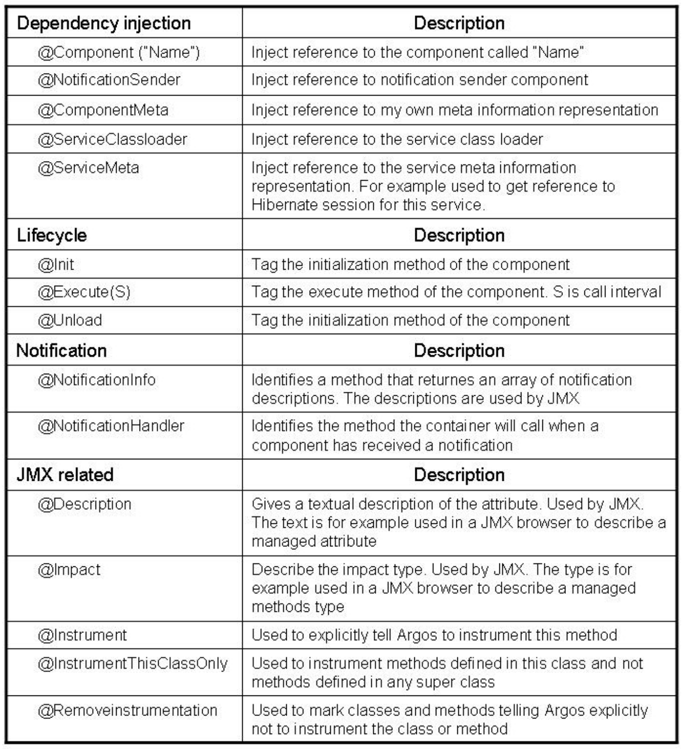
The annotations supported by the Argos core are listed in figure 2.

The annotations that are supported by the Argos core are as the figure shows divided into the following categories:

**Lifecycle** Annotations to control the component's lifecycle. They are used to annotate functionality performed in a timely manner (e.g. every 10th second), or when a component is created or destructed

**Notification** Annotations to describe notifications and to handle sending and receiving of notifications (e.g this function will be performed when the given notification is received).

**JMX related** Annotations to declare and describe instrumented methods and attributes.



| **Dependency injection** | **Description** |
|---|---|
| @Component ("Name") | Inject reference to the component called "Name" |
| @NotificationSender | Inject reference to notification sender component |
| @ComponentMeta | Inject reference to my own meta information representation |
| @ServiceClassloader | Inject reference to the service class loader |
| @ServiceMeta | Inject reference to the service meta information representation. For example used to get reference to Hibernate session for this service. |
| **Lifecycle** | **Description** |
| @Init | Tag the initialization method of the component |
| @Execute(S) | Tag the execute method of the component. S is call interval |
| @Unload | Tag the initialization method of the component |
| **Notification** | **Description** |
| @NotificationInfo | Identifies a method that returnes an array of notification descriptions. The descriptions are used by JMX |
| @NotificationHandler | Identifies the method the container will call when a component has received a notification |
| **JMX related** | **Description** |
| @Description | Gives a textual description of the attribute. Used by JMX. The text is for example used in a JMX browser to describe a managed attribute |
| @Impact | Describe the impact type. Used by JMX. The type is for example used in a JMX browser to describe a managed methods type |
| @Instrument | Used to explicitly tell Argos to instrument this method |
| @InstrumentThisClassOnly | Used to instrument methods defined in this class and not methods defined in any super class |
| @Removeinstrumentation | Used to mark classes and methods telling Argos explicitly not to instrument the class or method |

**Fig. 2.** Argos core annotations

**Injection** Annotations for dependency injection (to get access to core and system service infrastructure). Dependency injection is a design pattern that decouples the client component from the system service implementation component [9, 10]

Annotations in Argos are standard Java annotations. Using an annotation does not directly affect a component's semantics, but they do affect the way an Argos component is treated by the Argos container and the deployed system services, which can in turn affect the semantics of the instantiated component. Annotations in Argos are inspected reflectively at deploy time by the Argos core and by all the previously deployed system services.

The following example shows how the *Weather* service POJO component uses the @Init and @Execute annotations:

```
public class Weather {
  // Instruct container that this is the Weather components init method
  @Init public void init() {
    ...
  }

  // Instruct container to call this method at 30s intervals
  @Execute(30) public void execute() {
    ...
  }
}
```

The Argos core uses reflection to identify the annotations supported by the Argos core and handles them appropriately when the service is deployed to the container. All annotations are handled before the component is started.

## 3 Argos deployment

One of the main functions of the Argos core is to provide the ability to deploy system services and user applications. System services and applications must be presented to Argos in a Java archive file (jar) and the content of the jar file must follow the Argos deployment specification. Basically, the specification defines that deployment meta information needs to be included in a separate deployment descriptor (deploy.xml file) and that service content that is not POJO components needs to be added in designated folders in the jar file. In the deployment descriptor it is possible to express the following:

- Service name and version
- Service dependencies – references to other services that have to be deployed for this service to work properly
- List of components that the service contains (including possibility to express instantiating of multiple instances of the same POJO)
  - Component dependencies
  - Listen to properties, i.e. other components this component receives notifications from
  - Attribute configuration, i.e. start values for configurable attributes

Argos will store the meta information associated with a service in core objects. This meta information is available to system services using dependency injection annotations. Before any components are instantiated, all service dependencies are checked and validated, errors are logged and the service is not started if the dependencies are not met. The following example shows the deployment descriptor for the web service system service[2]:

```
<service name="!!Webservices" version="1.0">
  <depend on="!!Jetty6"/>
  <deploy>
    <component name="!!XML-RPC" class="argos.bangbang.xmlrpc.XmlRpc">
      <listen to="ComponentManager" />
    </component>
    <component name="!!Axis" class="argos.bangbang.axis.Axis">
      <listen to="ComponentManager" />
    </component>
  </deploy>
</service>
```

[2] System services in Argos is always deployed in jar files with names that starts with two exclamation marks (!!). This has also led to the convention that system service names starts with two exclamation marks.

System services (as the web service) can listen to notifications emitted by the Argos core (in this case, the core component *ComponentManager*). In this way they can perform their necessary actions associated to events emitted by the core. The web service system service will for example inspect all deployed components for @Webmethod annotations when the core emits the *SERVICE_STARTED* notification. If such annotations are found, their associated methods will be added as callable web methods (i.e. accessible through XML-RPC and SOAP calls).

## 4 System services

A system service is a collection of components that augment the intrinsic functionality of the Argos core. System services are the pinnacle of Argos elasticity. Typically a system service consists of definitions for new annotations and components that semantically handle the functionality related to the new annotations. The web services system service is a good example. The general idea for this service is to make it easy for an application programmer to express that he wants to create a web service. The easiest way to express this would be to allow the application programmer to simply tag a method in a component's code in order to specify that the method shall be offered as a web method (similar to the WebMethod attribute in C#). It should be possible to use both XML-RPC and SOAP to invoke the newly created web method. The Argos web service, @Webmethod annotation allows the application programmer to do just that. If you tag a method in a component that is part of a service with the @Webmethod annotation, that method is automatically exposed as a web method by the Argos container (i.e. accessible through XML-RPC and SOAP calls).

The web service system service (!!Webservice) first defines the new annotation (@Webmethod). Secondly, it contains the necessary code to reflectively find all @Webmethod annotations in subsequently deployed user components. Thirdly, it interacts with the Argos core (using notifications and dependency injection) and the Jetty web server system service to create the glue between incoming web service calls and the appropriate method in the instantiated component. Figure 3 illustrates the concept. The figure shows the chain of events when a user application (UserApplication.jar) that depends on the web service and Jetty system services is deployed to the container. First (1) the Argos core emits the SERVICE_STARTED notification. The notification is received by the web service system service and it will use reflection (2) to find all (if any) @Webmethod annotations in all the deployed POJOs in the newly deployed user application. When it finds @Webmethod annotations it registers the web method with the Jetty system service (3). The web service system service uses dependency injection to access and update meta information about the user application.

Using annotations, notifications, reflection and dependency injection in this manner allows for dynamic deployable extensions to the Argos core.

The possibility to add system services makes it possible to create Argos configurations that fit specific needs. Although Argos in many ways resembles

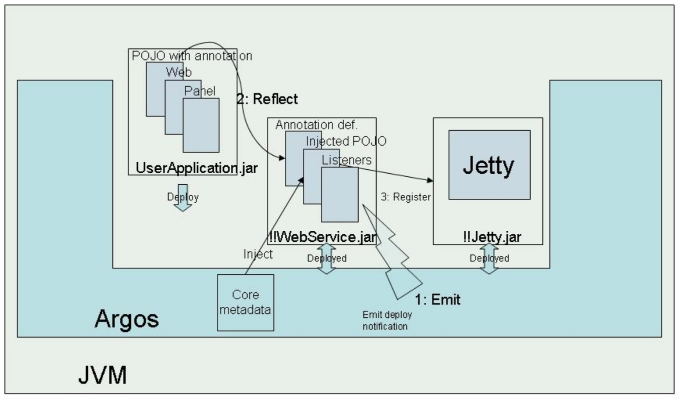


**Fig. 3.** Argos System Services

Enterprise Containers, the focus has been different. Enterprise containers need to be able to handle thousands of simultaneous requests to services that circle around legacy data. Our focus has been to provide a similar programming model for the creation of context aware, embedded or user-centric services that will be used by one or a handful of users. The design of Argos reflects that services are to be provided to a small number of simultaneous users. Argos allows deployed services to access the file system, open incoming and outgoing socket connections and to create threads. Enterprise systems usually do not allow such actions since it will make it extremely difficult to handle scaling and safety in a sensible manner (open connections to sockets or files and multiple threads makes it difficult to put components in a waiting pool). Argos components can with no restrictions behave in the same manner as POJOs running directly in a JVM. Argos adds expressive and powerful annotations to handle complicated tasks that are often needed when creating new services.

## 5 System service examples

System services can come in many shades. In the following subsections we will briefly describe a handful of system services that we have created at the time of writing.

The typical user application we have created uses sensors of some kind to gather context information. Sensors are often connected to a user's mobile phone. Typical sensors we have used are GPS, Step counters, temperature sensors, etc. In order to make it easy to use information from sensors in an Argos user application we have created a SMS system service, a Sensor system service and a TCP system service. The SMS service makes it possible to send and receive SMS message from a user application in Argos. The Sensor system service makes it possible to automatically connect new sensors to a users mobile phone (it also uses the SMS service for initial interaction with the phone's sensor framework client).

You will of course only deploy system services that your user application needs. In this way it is possible to create lean middleware support only for the features needed for your user applications. The result is an embedded or personal middleware system that suits your needs.

## 5.1 Sensor framework system service

The Argos Sensor framework actually consists of three separate parts. The first part is the Argos sensor system service which is deployed to the Argos container. The second part is a sensor configuration tool. The sensor system service handles incoming announcement requests from the sensor configuration tool. The sensor configuration tool is a standalone Java graphical tool used to describe the characteristics of a sensor. The description given by the user is transformed into a sensor configuration expressed as an XML document. When a sensor "announcement" is done from the tool, this description is transferred to the Argos Sensor framework system service. The sensor framework system service acts on the newly announced sensor configurations and handles all the sensor management needed to configure and start the sensor at the remote sensor location (i.e. connecting to the sensor, reading sensor data and transferring results back to the sensor framework system service. The third part of the sensor framework is a sensor host (remote) client program. The client program interacts with the Argos sensor system service to get the sensor configuration and download sensor specific plugins. The client program also automatically configures and starts the sensor and sends sensor data back to the Argos sensor system service as XML documents according to the Argos sensor data XML format. Currently we have only implemented a Windows Mobile 5.0 client program and a Java Windows client program. The Windows Mobile 5.0 client allows mobile phones with this operating system to act as Argos sensor framework clients. The Java Windows Client program allows Windows PCs to act as sensor framework clients (for example to manage USB or RS232 connected sensors). The typical scenario in the mobile setting is to connect Bluetooth or IR sensors to the mobile phone and send the collected sensor data to Argos using either a web service or TCP/IP interface. Once the Windows Mobile client program has been installed on the mobile phone, arbitrary Bluetooth or IR sensor may be connected to the phone. The installation, configuration and management of new sensors are done without touching the phone itself (i.e. remote management) [11].

## 5.2 TCP system service

The Argos TCP/IP system service provides a tcp/ip communication abstraction for Argos user applications. It utilizes a scalable architecture based on the Java Non-Blocking IO libraries in order to provide a high performance connectivity framework that can support at least hundreds of simultaneous tcp/ip connections. The service is primarily suited for Argos user applications that need server functionality, but it also provides tcp/ip client connectivity. The client and server interface that is exposed to Argos applications is identical and based on a stream abstraction. Argos applications either read from the stream or write to the stream after a tcp/ip connection is established. The following illustrates usage of this service.

```
// Inject handler for the tcpService
@Component("!!TCPBinding") public TCPBinding tcpService;

// Use handler to create a server, connectionAccepted is called when clients connects
tcpService.addService(PORT, new ClientHandler());

// When clients connects, this method in the components ClientHandler interface
// is called. ClientHandler is an interface that is implemented by the component
// Note: read is blocking, write is non blocking

public void connectionAccepted(Connection con){
   TCPStream stream = con.getStream();
   byte[] buf = new byte[10];
   stream.read(buf);
   stream.write(buf);
}
```

## 5.3 Small Messaging Service (SMS) system service

This service provides an easy to use abstraction (API) to send and receive SMS messages to/from mobile phones from Argos services. The following example shows how this system service is used to send a SMS message:

```
// Inject reference to SMS system service
@Component("!!SMSservice") public SMSservice smservice;
...
// Use the SMS system service to send an SMS
// The parameters are the phone number and the message
smservice.sendSMS("90914546", "How are you?");
...
```

This system service does not introduce any annotations. As the example shows, user applications can use dependency injection annotation, defined by the Argos core, to get a handler to the SMS system service component. This handler can then be used to send and receive SMS messages by invoking methods in the system service component.

## 5.4 Web system services

Currently we have developed three web related system components. The web service system service component has already been described. A web server (Jetty) is also deployed as a system service in Argos and in addition, support for Axis (SOAP) is included as a separate system service component (in the web service system service). The following examples shows how a user application component creates a web method. The web method can be reached using SOAP or XML-RPC when the component is deployed in the Argos container.

```
public class Something {
  @WebMethod public String hello() {
    return "Hello, world!";
  }
}
```

## 5.5 TwoWay notification system service

The *TwoWay system service* is a service that makes it easy to establish two way notification listening relationships, meaning that a service A listens to events submitted from service B and vice versa. The TwoWay system service lets either end establish the two way connection and avoids the problems of synchronizing the establishment of a two way connection between the two ends. This system service is useful when creating distributed services where user applications in different Argos containers need to cooperate. The following example shows how this system service is used to set up two way notification listening:

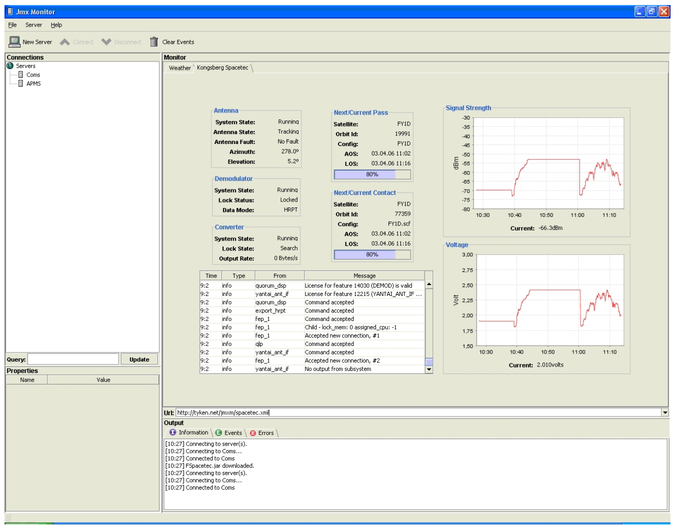
```
@Component("!!TwoWay") public TwoWay twoWay; // Inject reference

@ComponentMeta public ComponentMetaInfo meta; // Inject own meta info from core

@Init public void init() {
  ...
  // Set up mutual notification listening with the Manager component at host
  twoWay.setupTwoWay("Components:name=Manager", host, meta.getMyName());
  ...
}
```

## 5.6 Derby and Hibernate system services

Support for persistence is handled by two system services components, the Derby and Hibernate system service components. Derby is an SQL database and Hibernate is an object to entity relationship transformation tool. We have also experimented with a separate persistence system service that makes it possible to use annotations to express that component attributes are to be stored in the embedded database. We are currently considering using *Hibernate Annotations* instead.

## 5.7 JMX connector system service

The *JMX Connector system service* opens an RMI port to the Argos MBean server. When the RMI port is open, the JMX Connector system service makes it possible for remote monitoring and management of the Argos container and its deployed system services and applications. We have also developed a JMX browser and service monitoring and management tool specific for the Argos container but any JMX compliant monitoring and management tool may be used. The special thing about the JMX browser (called Argus) is that it can use service specific instrument panels to give advanced (graphical) insight into Argos and its system services and user applications. Figure 4 shows Argus in use. The left side shows ordinary JMX browsing and the right side shows a user application specific instrument panel for the satellite ground station service. The telemetry input from a satellite ground station is visualized in a user application specific instrument panel. The satellite ground station monitoring service is not discussed in this paper.



**Antenna**

| | |
|---|---|
| System State: | Running |
| Antenna State: | Tracking |
| Antenna Fault: | No Fault |
| Azimuth: | 278.0º |
| Elevation: | 5.2º |

**Demodulator**

| | |
|---|---|
| System State: | Running |
| Lock Status: | Locked |
| Data Mode: | HRPT |

**Converter**

| | |
|---|---|
| System State: | Running |
| Lock State: | Search |
| Output Rate: | 0 Bytes/s |

**Next/Current Pass**

| | |
|---|---|
| Satellite: | FY1D |
| Orbit Id: | 19991 |
| Config: | FY1D |
| AOS: | 03.04.06 11:02 |
| LOS: | 03.04.06 11:16 |

80%

**Next/Current Contact**

| | |
|---|---|
| Satellite: | FY1D |
| Orbit Id: | 77359 |
| Config: | FY1D.scf |
| AOS: | 03.04.06 11:02 |
| LOS: | 03.04.06 11:16 |

80%

Current: -66.3dBm

Current: 2.010volts

| Time | Type | From | Message |
|---|---|---|---|
| 9:2 | info | quorum_dsp | License for feature 14030 (DEMOD) is valid |
| 9:2 | info | yantai_ant_if | License for feature 12215 (YANTAI_ANT_IF ... |
| 9:2 | info | quorum_dsp | Command accepted |
| 9:2 | info | export_hrpt | Command accepted |
| 9:2 | info | fep_1 | Command accepted |
| 9:2 | info | fep_1 | Child - lock_mem: 0 assigned_cpu: -1 |
| 9:2 | info | fep_1 | Accepted new connection, #1 |
| 9:2 | info | qlp | Command accepted |
| 9:2 | info | yantai_ant_if | Command accepted |
| 9:2 | info | fep_1 | Accepted new connection, #2 |
| 9:2 | info | yantai_ant_if | No output from subsystem |

Url: http://tyken.net/jmxm/spacetec.xml

```
[10:27] Connecting to server(s).
[10:27] Connecting to Coms...
[10:27] Connected to Coms
[10:27] FSpacetec.jar downloaded.
[10:27] Connecting to server(s).
[10:27] Connecting to Coms...
[10:27] Connected to Coms
```

**Fig. 4.** Argus JMX monitoring and control tool

## 5.8 Service management and distribution system service

The *service management and distribution system service* is a framework to provide distribution of system services and user applications in Argos. A service provider creates a remote repository for their services and applications. These services and applications then become available to their end users through the service management and distribution system service. End users can download new services and applications or update their existing services and applications when an update is available.

The service management system service provides help in managing the container its deployed system services and applications. The end user can inspect meta data associated with each service, and configure these to fit their own needs. In addition, service management offers start, stop and updating (using service distribution) of services and applications.

## 5.9 Transaction management system service

An experimental transaction system service for Argos has been developed. The transaction system service supports flexible transaction processing by providing the possibility to support an extensible number of transaction managers. The current version of the experimental transaction system service uses two concurrently running transaction managers (DB-TM and WS-TM). The DB-TM (Database Transaction Manager) supports traditional ACID transactions implementing a two-phase commit protocol. The WS-TM (Web Service Transaction Manager) supports long-running transactions with relaxed atomicity following a compensation-based scheme. Based on the requirements from the application, one of them is selected to control the execution of an issued transaction. The Transaction Layer integrates both database sources and Web Services by implementing an abstraction layer facilitating the registration of, and the access to, the various sources.

## 6 Implementation details

The Argos core has been developed in Java 5. All functionality in the Argos core is realized as a set of JMX DynamicMBeans [12]. JMX defines an architecture for management of distributed resources (local or remote). Resources must be instrumented to be manageable. In Argos, the instrumentation is done by associating MBeans to resources. A very good overview of JMX related to development of middleware containers is found in section 2 of [13].

The JMX technology also provides a component-based architecture that makes it easier to develop a monitored and manageable middleware system (as compared to starting from scratch). Argos extends the JMX component model with elements related to component and service metamodels, component lifecycle and component dependency handling.

The Argos core instantiates all system services and user applications with a DynamicMBean proxy associated to them. This means that all the major functionality in the Argos core, all Java classes in system services and all Java classes in user applications are instantiated with associated DynamicMBean proxies. This makes it possible to monitor and manage the Argos core, the system services and user applications through JMX.

System service or user application programmers are not exposed to JMX or MBeans, meaning that Java classes in system services and user applications does not have to implement any of the MBean interfaces. The component model exposed to application programmers is plain Java (with the possibility to add Argos core and system service specific annotations). Using the proxy pattern together with reflection and explicitly expressed meta-data is very powerful as it makes it possible to turn any java object into a DynamicMBean at run-time. The reflective inspection done by the DynamicProxy is the first of a series of reflective inspections performed on newly deployed components. The Argos core performs another inspection just prior to activating the new component. This run is done to collect information given by lifecycle annotations. In addition, potentially all deployed system services may inspect every newly deployed component to search for annotations that are part of that system services supported annotations (if any). The system service will also perform the actions (service) associated with the annotation. Dependency injection is just a special type of annotation and may be handled in all the reflective inspection passes (depending of what you would like to inject). Figure 5 illustrates the situation when a POJO has been deployed to the Argos container.

Since JMX and specifically MBeans are intentionally not exposed to the application program, the Argos container creates a dynamic MBean acting as

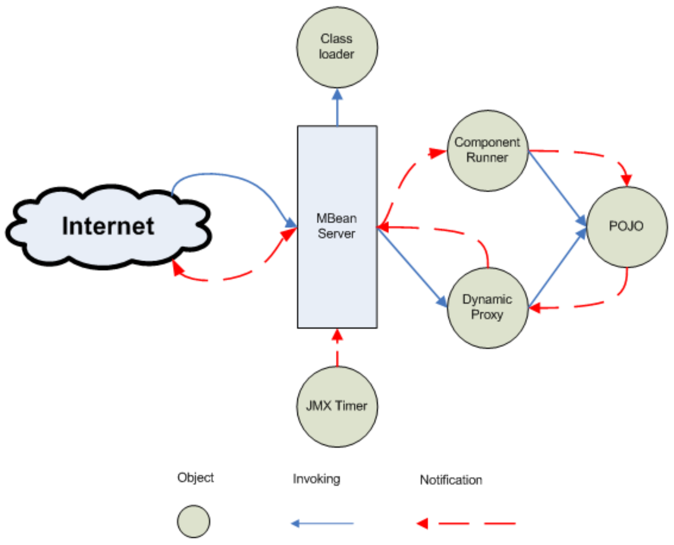


**Fig. 5.** MBean server, DynamicProxy and POJO relationship

a proxy to the POJO. The proxy MBean and the Component runner MBean handle all interaction between the MBean server and the POJO.

## 7 Example applications

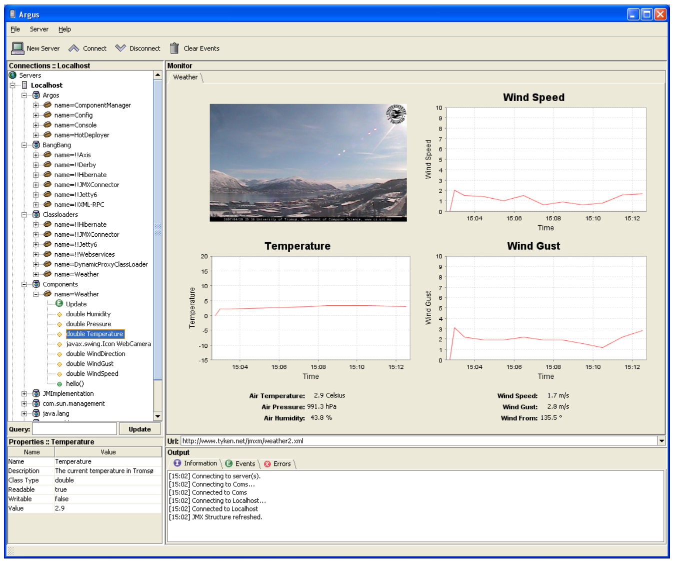
The Argos container has evolved as a result of needs/requirements that we have observed when developing software in the areas of:

- Applications related to sensor networks
- Context aware applications including usage of several types of sensors as context information sources
- Personal or small scale deployed services, i.e. services to one or a small number of persons
- Personalized services in a mobile phone setting, i.e. applications available using the mobile phone as end user terminal and sensors related to a person

Many of these applications follow the basic Input, Processing, Output pattern. The Sensor framework system services has been developed for Argos to make it easy to include sensors of different types as input sources for Argos applications. Currently, processing is usually performed in processing components, but we have ongoing work to include rule based processing as a separate system service. Output can easily be done to a database or to external endpoints through web services. Visualization of output can be done through GUI, Widgets or instrument panels. Some of the user applications that we have developed are briefly explained in the subsequent sections.

## 7.1 The Weather Service

This service consists of a very simple component that reads sensor data from weather instruments. The data is visualized in a service specific instrument panel. The instrument panel (viewed in the Argos JMX monitor) is shown in figure 6.



Connections :: Localhost

```
Servers
  Localhost
    Argos
      name=ComponentManager
      name=Config
      name=Console
      name=HotDeployer
    BangBang
      name=!!Axis
      name=!!Derby
      name=!!Hibernate
      name=!!JMXConnector
      name=!!Jetty6
      name=!!XML-RPC
    Classloaders
      name=!!Hibernate
      name=!!JMXConnector
      name=!!Jetty6
      name=!!Webservices
      name=DynamicProxyClassLoader
      name=Weather
    Components
      name=Weather
        Update
        double Humidity
        double Pressure
        double Temperature
        javax.swing.Icon WebCamera
        double WindDirection
        double WindGust
        double WindSpeed
        hello()
    JMImplementation
    com.sun.management
    java.lang
```

Query:

Properties :: Temperature

| Name | Value |
|---|---|
| Name | Temperature |
| Description | The current temperature in Tromsø |
| Class Type | double |
| Readable | true |
| Writable | false |
| Value | 2.9 |

**Air Temperature:** 2.9 Celsius
**Air Pressure:** 991.3 hPa
**Air Humidity:** 43.8 %

**Wind Speed:** 1.7 m/s
**Wind Gust:** 2.8 m/s
**Wind From:** 135.5 °

Url: http://www.tyken.net/jmxm/weather2.xml

```
[15:02] Connecting to server(s).
[15:02] Connecting to Coms...
[15:02] Connected to Coms
[15:02] Connecting to Localhost...
[15:02] Connected to Localhost
[15:02] JMX Structure refreshed.
```

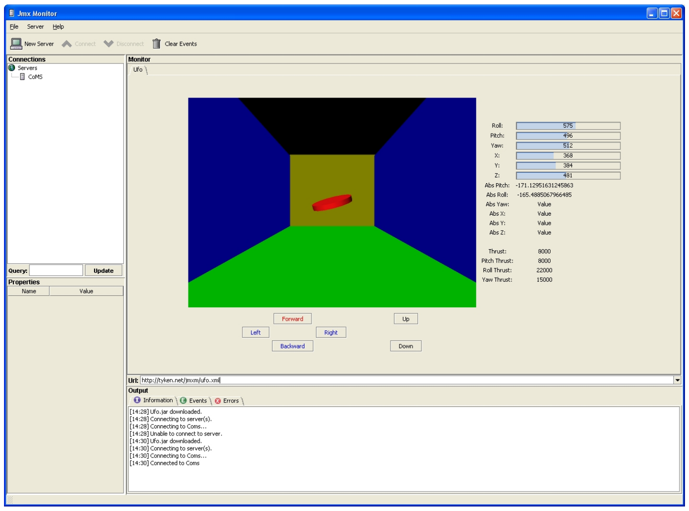
**Fig. 6.** The Weather service instrument panel

The weather service uses the *TCP system service* to bind to external weather sensors. The sensor values are stored in a database using the *Hibernate and Derby system service*. The service specific instrument panel is the only GUI for this service and the *JMX Connector system service* is used to make it possible to remotely connect to the Argos container that runs the weather service.

## 7.2 The XUfo Service

Automatically piloting of a flying radio controlled helicopter (called XUfo) using Bluetooth accellerometer and gyro sensors. The Argos components in this service read sensor data approximately every 10 ms, use Kalman filters [14] to adjust the readings and then compute control signals which are transferred back to the helicopter. The service also includes an instrument panel that visualizes the helicopter in a virtual 3D room. Figure 7 shows this instrument panel.

The XUfo service uses the *Sensor Framework system service* to connect to the Bluetooth sensor package on the helicopter (through a USB Bluetooth dongle). It



```
Roll:          575
Pitch:         496
Yaw:           512
X:             368
Y:             384
Z:             481
Abs Pitch:     -171.12951631245863
Abs Roll:      -165.4885067966485
Abs Yaw:       Value
Abs X:         Value
Abs Y:         Value
Abs Z:         Value

Thrust:        8000
Pitch Thrust:  8000
Roll Thrust:   22000
Yaw Thrust:    15000
```

Url: http://tyken.net/jmxm/ufo.xml

```
[14:28] Ufo.jar downloaded.
[14:28] Connecting to server(s).
[14:28] Connecting to Coms...
[14:28] Unable to connect to server.
[14:30] Ufo.jar downloaded.
[14:30] Connecting to server(s).
[14:30] Connecting to Coms...
[14:30] Connected to Coms
```

**Fig. 7.** The XUfo service instrument panel

also uses the Argos core notification annotations to bind the input and processing POJO components together (the input POJO emits notifications when new sensor readings are available).

## 7.3 Lifestyle Services

Together with the Norwegian Center for Telemedicine we are developing a set of services that we have called "Lifestyle Services". In these services we are using sensors to read end user biometrics (for example Blood Glucose level, heart rate, activity level etc.). The service uses information from the sensors combined with a user profile and input from the end user to utilize behavioral change mechanisms in order to try to affect the end user's lifestyle. The sensors are connected to the end user's mobile phone. The Argos *Sensor Framework System Service* is used to connect sensors and to configure the transmission of sensor data from the mobile phone to the Argos container. Data from sensors is stored in a database using the *Hibernate and Derby system services* and is further processed using rules and processing components to calculate the interaction with the end user in order to attempt to change the end user's behavior.

## 7.4 Experience Sampling Service

The method called Experience Sampling Method(ESM) [15] aims at capturing immediate experiences from participants in a survey. Combining ESM with mobile technology gives the opportunity to design surveys that are to capture immediate experiences.

Using the features of Argos, a software tool for generating ESM based surveys has been designed and implemented. This tool, named esmDesk, is deployed in Argos as a user application and provides an experimenter with a graphical user interface where ESM based surveys can be created, modified and distributed to a set of mobile devices. Distribution of surveys, which are expressed in XML, is done using web services. When the experimenter has finished creating the survey using esmDesk, a set of participants is selected and esmDesk notifies the participants' mobile phones by sending an SMS message. The platform for running surveys on the mobile phone is called esmMobile and it receives the SMS message notifying that a new survey is available. Using web services esmMobile downloads the XML representation of the survey from esmDesk, interprets it and starts running the survey. After all the elements of the survey have been answered by the participant, answers are sent back to esmDesk, again using web services. The results are store in a database using Argos persistence support.

### 7.5 Others

Some other experimental services developed using the Argos middleware or some of the preceding versions of Argos is very briefly presented here:

**Herding** The electronic shepherd system provides farmers with information describing the state of their animals. The information collected was used by a back-end system and generated various map views with associated animal alarms. GPS, temperature and motion sensors were used.

**YPIV** Your Personal Infotainment Vault service will serve content to, for example, your mobile phone and to others that you decide to share content with. The content you manage in your YPIV can for example range from: Images, Music, Movies, Context information, PIM services, Ring tones, MMS content, Documents etc. The ”sensors” in this service were radio station rippers and other content ingestion ”sensors”.

**FiFamos** The main problem in a fish farm is that most of them are without supervision for a long time while they are exposed to changing weather conditions. FiFaMos is an advanced surveillance and alarm system for sea farms. GPS, camera, temperature, wind, wave, current, water quality and food level sensors were used.

## 8 Evaluation

Argos is evaluated by demonstrating that it matches the needs of the target application domains and by comparing it with other related projects.

### 8.1 Usage

In section 7 we have described some of the applications developed using Argos. Argos (including earlier versions of Argos called COMS and APMS) has been used to develop demonstrators internally at the lab and with external partners. Currently several projects of external partners have decided to use Argos in both research projects and in the development of prototypes and demonstrators of new services and products (including Norwegian Center of Telemedicine and Telenor R&I, Telenor is a Norwegian telecom company).

The core functionality of the Argos core (service and component model, lifecycle, notification, instrumentation and dependency injection) and functionality provided by deployed system services makes it easier to quickly develop new applications. The programmer can focus on the core functionality of the application.

The Weather application, the XUfo application and the Lifestyle applications described in section 7 all show how easy it is to integrate sensor data in an Argos application. The Argos core can be easily extended with new system services, and the Sensor framework system service (see section 5.1) matches some of the needs of these applications perfectly. In the Weather service the weather sensor data are collected, stored (see persistence provided by the Hibernate system service in section 5.6), and presented (see JMX Connector system service in section 5.7) with little effort from the application developer.

The XUfo application demonstrates that Argos can also be used in a (near real-time) control system. The efficiency of the Argos core and its notification support makes it possible to pilot a flying object with gyro and accellerometer sensors and a feedback loop through the Argos core that includes processing, visualization, and control signal computation (see figure 7).

The Experience Sampling application uses the persistence support provided by the Hibernate system service to easily store data collected from several respondents. The requirements of the Experience Sampling application also resulted in a new system service for SMS (Small Messaging Service). This system service are used in completely different applications that also includes mobile phones and SMS messaging (i.e. the Lifestyle service).

## 8.2 Related work

Prism-MW [16] defines its setting as "programming-in-the-small-and-many". They claim to have a flexible, efficient, scalable and extensible platform for this setting of small, resource constrained, and highly mobile computing platforms. Flexibility is achieved in a similar way as in Argos by providing a core including a component model and events. However, the actual platform is very different. Connectors are an important part of the Prism-MW core and their task is to route events. Each component can be attached to any numbers of connectors, and each connector can serve any numbers of components. This flexibility is also used for system reconfigurability. Their focus on scalability and efficiency are not found in Argos and Argos extensibility can therefore not be compared to Prism-MW. In Prism-MW extensibility is provided by extending the core programming model (extending the connector class, the component class or the event class). Argos is extended by deploying new system-services at run-time. Hot deployment makes it possible to extend (and update) Argos at run-time.

Argos provides more features for lifecycle support and system services and is aimed for different application domains.

JBoss [13] is a feature rich application server platform. Similar to Argos it has a core and is extended with system components to provide different system services. The difference here is that JBoss supports complex enterprise applications while Argos supports a completely different set of applications. Both the size (memory print and lines of codes) and the complexity of these two applications platforms differs a lot. We also argue that complexity for the application programmer is higher when using JBoss (or any other EJB application server) than using Argos. This is due to the focus on enterprise applications where more control is moved to the application server (obviously for efficiency and scaling, but also for safety and isolation). Another important difference is that in Argos, the programmer has access to local resources in the same way as any desktop application that is implemented using Java. This is important for many applications in the application domains targeted by Argos.

Gaia [17] and Mobile Gaia [18] is tied to the concept of Active Spaces where physical and computational infrastructure are merged into an integrated habitat. The focus is implicit support for resource awareness (discovery), multi-device interaction, context sensitivity, mobility, run-time adaption and user-centrism. This is very different from Argos. Argos could probably be used to develop this kind of platform by providing system services matching the functionality of Active Spaces. In Argos the term user-centric is used to describe applications and services accessing and using resources close to or related to the user (personal assets or private sensor data or similar). In Gaia this is used to describe the need of the application to adapt to the user (his context and preferences).

Another group of related platforms are MIDAS [19], JAGR [20], The Collective [21] and [22]. All of these platforms tries to solve the problem of (self) adaptive containers. In Argos it should be possible to add context aware system services that has similar approaches, but this is not part of Argos core.

## 9 Conclusions

The main contributions from the Argos project is that it gives useful (for this domain) enterprise container type support (e.g. component model, lifecycle support and persistence) to desktop and user-centric application development, without the complexity and limitations enforced by enterprise containers.

This grants developers of desktop and user-centric applications the advantage of tailored and advanced, flexible and extensible, middleware support. The result is the possibility to rapid develop feature rich applications that integrates and aggregates information from different sources and presents the results in different settings. Information sources can be sensors, user input, filesystem, databases, web services, and so on. The aggregated and processed data can in turn easily be presented as web pages, desktop widgets, web services, instrument panels or ordinary graphical user interfaces. The collected, aggregated and processed data can be easily persisted or propagated for further processing by other components.

The Argos middleware provides a leaner platform for development of desktop applications, demonstrators, prototypes and experimental middleware development than what would be the case if using for example the JBoss application container as a basis. The Argos core and default system services (Hibernate, Derby, Jetty, JMX Connector, web service) together counts 4,700 physical source code lines using SLOCCount[23]. In comparison, the JBoss microkernel alone counts 10,844 and a complete Jboss installation contains 630,443 physical source code lines using SLOCCount.

Currently the performance of Argos is beeing tested. Some preliminary tests have shown that the deployment time (including dependency testing) of new components grows linearly with the number of components deployed (tested up to 50,000 components). Tests have also shown that notifications are an efficient way to interact between components, Argos core, and system services. Dependency injection is not as efficient as notifications (probably because it uses reflection). The results have shown that usage of dependency injection takes three times longer than notifications. The most efficient way is ordinary method calls, but this differ from the two other approaches since it is not possible between components in different Argos containers.

The Argos middleware and Argus JMX monitor and all system services we have developed is stable and available under a BSD licence (open source software). More information about Argos can be found here:

`http://argos.cs.uit.no`

## References

1. Andersen, A., Blair, G., Goebel, V., Karlsen, R., Stabell-KulÃ¸, T., Yu, W.: Arctic beans: Configurable and re-configurable enterprise component architectures. IEEE Distributed Systems Online **2**(7) (2001)
2. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification. Third edition edn. ADDISON-WESLEY (2005)
3. Sun Microsystems: Java annotations: Jsr 175, a metadata facility for the java programming language: http://www.jcp.org/en/jsr/detail?id=175 (2004)
4. Jetty: a full-featured web server implemented entirely in java: `http://jetty.mortbay.org` (2007)
5. Hibernate: an object/relational persistence and query service: `http://hibernate.org` (2007)
6. Derby: a relational database implemented entirely in java: `http://db.apache.org/derby` (2006)
7. W3C: Simple object access protocol (soap) 1.1: `http://www.w3.org/TR/soap` (2000)
8. XML-RPC: remote procedure calling using http as the transport and xml as the encoding: `http://www.xmlrpc.com` (1998)
9. Fowler, M.: Inversion of control containers and the dependency injection pattern: `http://www.martinfowler.com/articles/injection.html` (2004)
10. Fowler, M.: Module assembly [programming]. Software, IEEE **21**(2) (2004) 65–67
11. OMA: Mobile device management protocols and mechanisms: `http://www.openmobilealliance.org/tech/wg\_committees/dm.html` (2007)

12. Sun Microsystems: Java management extensions: http://jcp.org/aboutJava/communityprocess/final/jsr003/index3.html (2006)
13. Fleury, M., Reverbel, F.: The jboss extensible server. In Schmidt, D., ed.: Middleware 2003 – ACM/IFIP/USENIX International Middleware Conference. Volume 2672., Springer-Verlag (2003) 344–373
14. Kalman, R.E.: A new approach to linear filtering and prediction problems. Transactions of the ASME - Journal of basic Engineering **82** (1960) 34–45
15. Conner, T.: Experience sampling resource page (2006) http://psychiatry.uchc.edu/faculty/files/conner/ESM.htm.
16. Mikic-Rakic-M, Medvidovic-N: Adaptable architectural middleware for programming-in-the-small-and-many. In Endler, M., Schmidt, D., eds.: Middleware 2003. ACM/IFIP/USENIX International Middleware Conference. Proceedings. Rio de Janeiro, Brazil. ACM. IFIP. Adv. Comput. Syst. Assoc. 16 20 June 2003., Springer-Verlag, Berlin, Germany (2003)
17. Roman, M., Campbell, R.: A middleware-based application framework for active space applications. In: ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil (2003)
18. Chetan, S., Al-Muhtadi, J., Campbell, R., Mickunas, M.D.: A middleware for enabling personal ubiquitous spaces. In: System Support for Ubiquitous Computing Workshop at the Sixth Annual Conference on Ubiquitous Computing, Nottingham, England (2004)
19. Popovici-A, Frei-A, Alonso-G: A proactive middleware platform for mobile computing. In Endler, M., Schmidt, D., eds.: Middleware 2003. ACM/IFIP/USENIX International Middleware Conference, Springer-Verlag, Berlin, Germany (2003)
20. Candea-G, Kiciman-E, Zhang-S, Keyani-P, Fox-A: Jagr: an autonomous self-recovering application server. In Parashar, M., Hariri, S., Raghavendra, C., eds.: AMS 2003, Autonomic Computing Workshop: 5th Annual International Workshop on Active Middleware. Seattle, WA, USA. IBM. Nat. Sci. Found. Soc. Modeling Simulation. IEEE. IEEE Comput. Soc. Arizona Center for Integrative Modeling Simulation. Univ. Southern California. WINLAB, Rutgers Univ. 25 June 2003., IEEE Comput. Soc, Los Alamitos, CA, USA (2003)
21. Edward, C., Enda, R.: The collective: a common information service for self-managed middleware. In: Proceedings of the 4th workshop on Reflective and adaptive middleware systems, Grenoble, France, ACM Press (2005) 1101528.
22. Gang-Huang, Tiancheng-Liu, Hong-Mei, Zizhan-Zheng, Zhao-Liu, Gang-Fan: Towards autonomic computing middleware via reflection. In: Proceedings of the 28th Annual International Computer Software and Applications Conference. COMPSAC 2004. Hong Kong, China. 28 30 Sept. 2004., IEEE Comput. Soc, Los Alamitos, CA, USA (2004)
23. Wheeler, D.: Sloccount, tools for counting physical source lines of code (sloc): http://www.dwheeler.com/sloccount (2007)