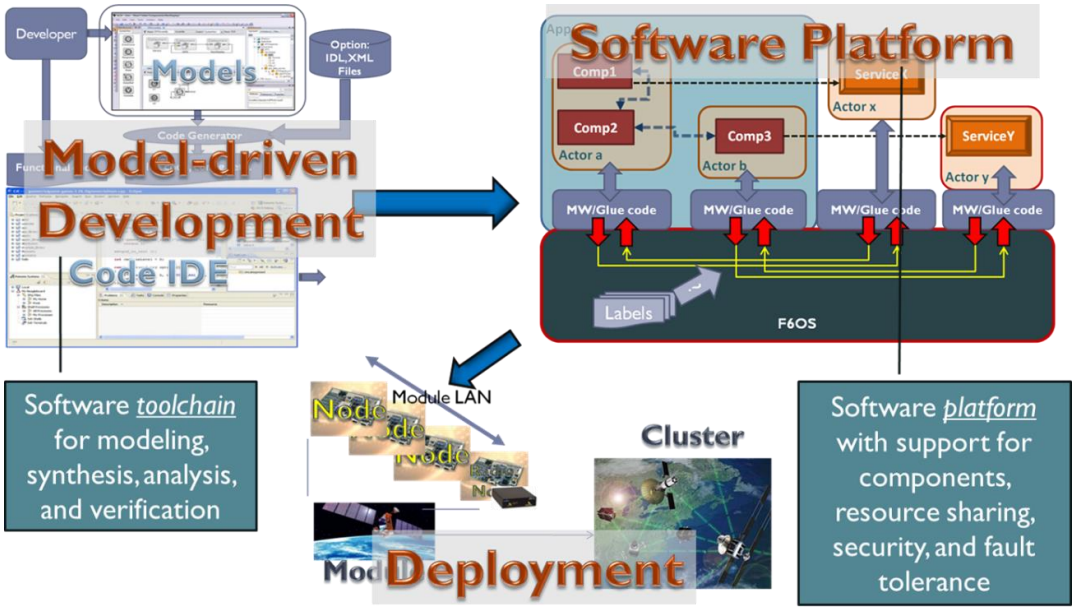
F6 Model-driven Development Kit: Information Architecture Platform

Last updated: June 12, 2013

The F6 IAP is a state-of-the-art software platform and toolsuite for the model-driven development of distributed real-time embedded systems[1]. It consists of two major subsystems: (1) a *design-time toolsuite* for modeling, analysis, synthesis, implementation, debugging, testing, and maintenance of application software built from reusable components, and (2) a *run-time software platform* for deploying, managing, and operating application software on a network of computing nodes. The platform is tailored towards a managed network of computers and distributed software applications running on that network: a cluster of networked nodes.

The toolsuite supports a model-based paradigm of software development for distributed, real-time, embedded systems, where modeling tools and generators automate the tedious parts of software development and also provide a design-time framework for the analysis of software systems. The run-time software platform reduces the complexity and increase reliability of software applications by providing reusable technological building blocks in the form of an operating system, middleware, and application management services.



The Information Architecture Platform

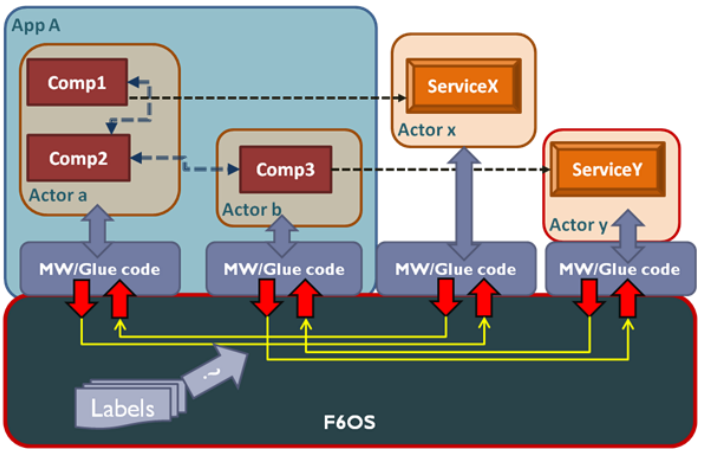
The IAP is a complete, end-to-end solution for software development: from modeling tools to code to deployed applications. It is open and extensible, and relies on open industry standards, well-tested functionality and high-performance tools. It focuses on the architectural issues of the software, and promotes the modeling of application software, where the models are directly used in the construction of the software.

[1] This work was supported by the DARPA System F6 Program under contract NNA11AC08C through NASA ARC.

Applications

Software applications running on the IAP are distributed: an application consists of one or more *actors* that run in parallel, typically on different nodes of a network. Actors specialize the concept of processes: they have identity with state, they can be migrated from node to node, and they are managed. Actors are created, deployed, configured, and managed by a special service of the run-time platform: the deployment manager – a privileged, distributed, and fault tolerant actor, present on each node of the system, that performs all management functions on application actors. An actor can also be assigned a limited set of resources of the node it runs on: memory and file space, a share of CPU time, and a share of the network bandwidth.

Applications are built from *software components* – hosted by actors – that interact via *only* well-defined interaction patterns using security-labeled messages, and are allowed to use a specific set of low-level services provided by the operating system. The low-level services include messaging and thread synchronization primitives, but components use these indirectly: via the middleware libraries.



Applications, actors, components, and services

The middleware libraries implement the high-level communication abstractions: synchronous and asynchronous interactions, on top of the low-level services provided underlying distributed hardware platform. Interaction patterns include (1) point-to-point interactions (in the form of synchronous and asynchronous remote method invocations), and (2) group communications (in the form of asynchronous publish-subscribe interactions). Component operations can be event-driven or time-triggered, enabling time-driven applications. Message exchanges via the low-level messaging services are time-stamped, thus message receivers are aware of when the message was sent. Hence temporal ordering of events can be established (assuming the clocks of the computing nodes are synchronized).

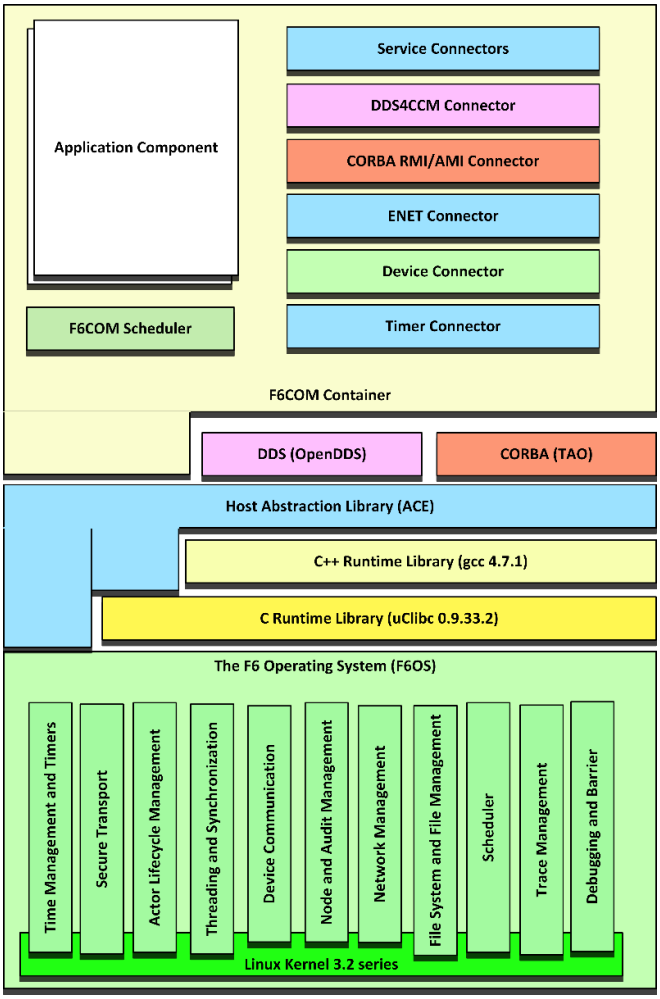
Specialized, verified *platform actors* provide system-wide high-level *services:* application deployment, fault management, controlled access to I/O devices, etc. Each application actor exposes the interface(s) of one or more of its components that the components of applications can interact with using the same interaction patterns. Applications can also interact with each other the same way: exposed interfaces and precisely defined interaction patterns.

The F6 Operating System – a set extension to the Linux kernel – implements all the critical low-level services to support resource sharing (incl. spatial and temporal partitioning), actor management, secure (labeled and managed) information flows, and fault tolerance. A key feature of the OS layer is support for temporal partitions (similarly to the ARINC-653 standard): actors can be assigned to a fixed duration, periodically repeating interval of the CPU's time so that they have a guaranteed access to the processor

in that interval. In other words, the actors can have an assured bandwidth to utilize the CPU and actors in separate temporal partitions cannot inadvertently interfere with each other via the CPU.

Run-time Software Platform

The run-time software platform has several layers, as shown in the figure. Practically all layers are based on existing and proven open-source technology. Starting from the bottom, the operating system layer extends the Linux kernel with a number of specific services, but it strongly relies on the code available in the Linux kernel (currently: version 3.2.17). The advantage of this approach is that developers can use existing Linux system calls, side-by-side with the F6OS system calls.

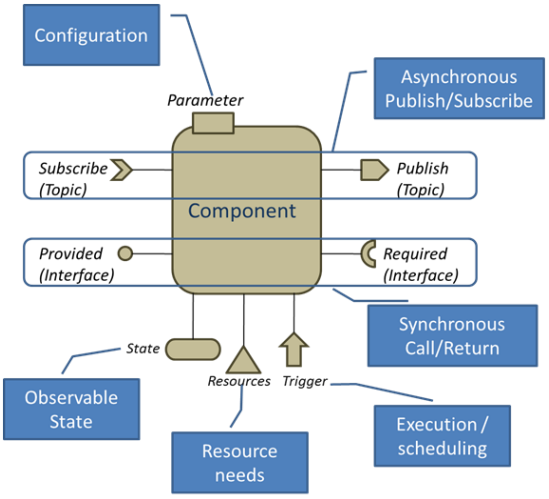


IAP Run-time Software Layers

The C and C++ run-time support libraries implement the conventional support services needed by the typical C and C++ programs. The C run-time library has entry points to access the F6OS system calls. These calls utilize data structures that have been defined using the standard Interface Definition Language (IDL), and can be created and manipulated using generated constructor and manipulation operators. The implementation of the F6OS operating system calls checks the integrity of all data structures passed on the interface. This enables validation of the data structures on the interface, preventing potential abuse of the F6OS system calls.

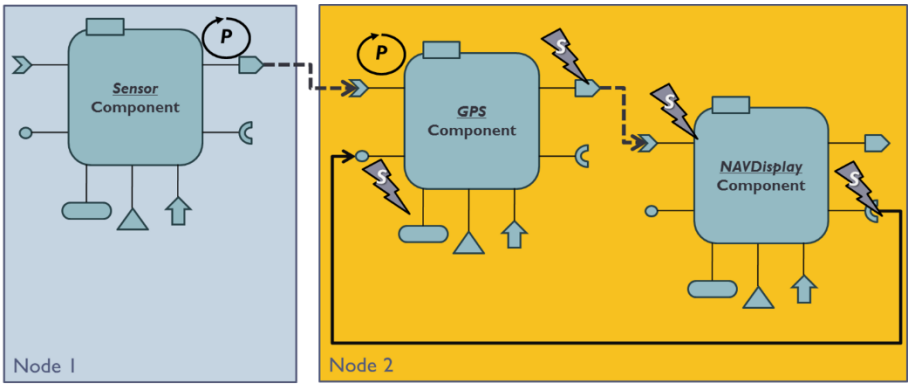
Layered on the C and C++ run-time libraries the Adaptive Communication Environment (ACE) libraries provide a low-overhead isolation layer for the higher level middleware elements that support CORBA and DDS. The CORBA implementation is based on The ACE ORB (TAO, currently: version 6.1.4) that implements a subset of the CORBA standard for facilitating point-to-point interactions between distributed objects. Such interactions are in the form of Remote Method Invocations (RMIs) or Asynchronous Method Invocations (AMIs). RMIs follow the call-return semantics, where the caller waits until the server responds, while the AMIs follow the call-return-callback semantics, where the caller continues immediately and the response from the server is handled by a registered callback operation of the client. The CORBA subset implemented by the middleware has been selected to support a minimal set of core functions that are suitable for resource-constrained embedded systems.

The DDS implementation is based on the OpenDDS (currently: version 3.4) that implements a subset of the DDS standard for facilitating anonymous publish/subscribe interactions among distributed objects. In these interactions publishers send typed messages of specific *topics* via the middleware which then distributes them to subscribers interested in those topics. Subscribers can be anywhere on the network, they can join and leave the system at any time – the distribution middleware decouples publishers from the subscribers. There are several quality-of-service attributes associated with publishers and subscribers that control features like buffering, reliability, delivery rate, etc. DDS is designed to be highly scalable, and its implementations meet the requirements of mission-critical applications.



A component

CORBA and DDS provide for data exchange and basic interactions between distributed objects, but in the IAP objects are packaged into higher-level units called *components*. A component, shown on the figure, *publishes* and *subscribes* to various topics (possibly many), implements (thus *provides*) interface(s), and expects (thus *requires*) implementations of interfaces. Note that a component may contain several, tightly coupled objects. Components may expose (part of) their observable state via read-only state variables, accessible through specific methods. Components are configured via parameters, and have memory resources needs. Component operations are scheduled based on events or elapse of time. An event can be the arrival of a message the component has subscribed to or an incoming request on a provided interface. Time triggering is done by associating a timer with the component that invokes a selected operation on the component when a configurable amount of time elapses, possibly periodically repeating the operation. Component operations can perform computations, publish messages, and call out to other components via the required interfaces. To avoid having to write complex locking code for components, component operations are always single threaded: inside of one component at most one thread can be active at any time.



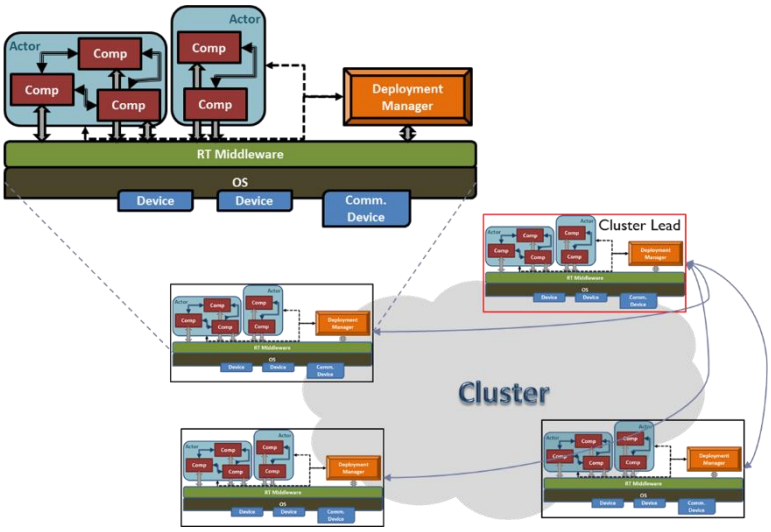
Interacting components deployed on two different nodes

Actors are formed from interacting components, and applications are formed from actors that interact with each other via their interacting components. Actors (together with their components) can be deployed on different nodes of a network, but their composition and interactions are always clearly defined: they must happen either via remote method invocations or via publish/subscribe interactions. The figure above shows an application where a

`Sensor` component periodically (P) publishes a message that a `GPS` component subscribes and which, in turn, sporadically (S) publishes another message that a `NAVDisplay` component consumes. This last component invokes the `GPS` component via a provided interface, when it needs to refresh its own state. The messages published can be quite small, while the method invocation (that happens less frequently, and on demand) may transfer larger amounts of data. The number of possible combination of interactions among components is quite large, but each interaction pattern is precisely defined, allowing the application writer to understand all operational scenarios. Note that applications are multi-threaded, only individual components are single threaded.

Interactions are realized by *connectors* that support specific interaction patterns. In addition to the two main ones described above, components can interact with network sockets (for conventional message oriented networking using POSIX standard socket APIs), timers, and I/O devices. For each case, the synchronization between component code execution and the events of the external world are precisely defined, and allow the implementation of various interactions to enable high degree of asynchrony and responsiveness.

The run-time software platform includes a key platform actor: the Deployment Manager (DM) that instantiates, configures, and dismantles applications. Every node on a network has a copy of the DM that acts as a controller for all applications on that node. The DMs communicate with each other, with one being the lead 'Cluster' DM. This, cluster leader DM orchestrates the deployment of applications across cluster with the help of the node DMs. For deployment the binaries of application components and a deployment plan (an XML file) should be placed on each node, then the cluster lead DM reads and executes the plan: it starts the actors, installs components, configures the network connections among the components, etc., and finally activates the components. This last step releases the execution threads of the components. When the applications need to be removed, the DM stops the components, removes the network configurations, and stops the actors. A key feature of the deployment process is that the network connections among the parts: i.e. actors and components of the distributed application are managed: the application business logic does not have to deal with this problem; everything is set up based on the deployment plan.

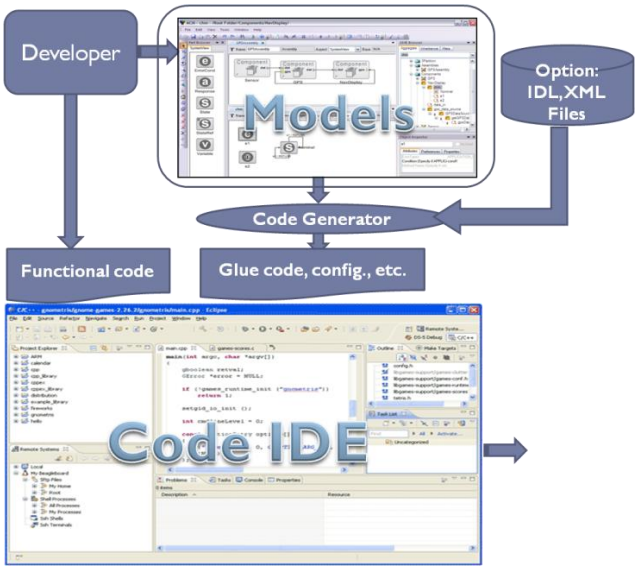


Deployment Manager and Applications

Design-time Development Platform

Configuring the middleware and writing code that takes advantage of the component framework is a highly non-trivial and tedious task. To mitigate this problem and to enable programmer productivity a model-driven development environment is available that simplifies the tasks of the application developers and system integrators.

In this environment, developers define with graphical and textual models various properties of the application, including: interface and message types, components types (in terms interfaces and publish/subscribe message types), component implementations, component assemblies, and applications (in terms interacting components and actors containing them). Additionally, the hardware platform for the cluster can be modeled: processors, network and device interfaces, network addresses, etc. Finally, the deployment of the application(s) on the hardware platform can be modeled (in terms mapping actors onto hardware nodes, and information flows onto network links). The deployment can change over time but the deployment is stable and static for most of the time. Models are processed by code generators that produce several artifacts from them: source code, configuration files, scripts that facilitate the automated compilation and linking of the components, and other documents. The application developer is expected to provide the component implementation code in the form of C++ code (currently; in the future: any other, supported executable language) and add it to the generated code. The compilation and debugging of the applications can happen with the help of a conventional Interactive Development Environment (currently: Eclipse) that supports editing, compiling, and debugging the code. The result of this process is a set of component executables and the deployment plan – ready to be deployed on a cluster of nodes.



Model-driven Development

The model-driven approach has several benefits. (1) The model serves as the single source of all structural and configuration information for the system. (2) The tedious work of crafting middleware 'glue' code and configuration files for deployment is automated: everything is derived programmatically from the models. (3) The models provide an explicit representation of the architecture of all the applications running on the system – this enables architectural and performance analysis on the system before it is executed. (4) Models can also be used for rapidly creating 'mockup' components and applications for rapid prototyping and evaluation.

Summary

The F6IAP provides a sophisticated, end-to-end solution for building and running distributed real-time embedded applications. It contains not only a run-time framework with a state-of-the-art operating systems extended with special features for resource, application, and network management together with a component framework with a precise defined model of computation, but also a model-driven development toolchain that assists developers and integrators in managing the development process.