A lightweight, code generated and fast IPC-framework for C++ based applications

Martin Kalisch, Senior Software Architect, Continental Automotive
Peter Reitinger, Software architect, Continental Automotive
Stefan Bitzer, Software architect, Continental Automotive
Valentin Uritescu, Software architect, Continental Automotive

Villingen, December 2011

Introduction

The increasing complexity of embedded products introduced the need for the extensive use of the multiprogramming concept. For modern software systems, it is essential to have systems for communication and synchronization management between cooperating processes. Inter-Process-Communication (IPC) provides the solution by coordinating the activities the of cooperating processes.

Continental's tolling and telematic products are based on an architecture that separates the hardware abstraction and the application (functionality) process-spaces because of security-requirements and loose-coupling-requirements and in order to have several separately qualified applications running on one machine at the same time without interfering with each other. Continental's LEAP (Linux Embedded Automotive Platform) offers the hardware-abstraction for these products in the Commercial-Vehicle-division. LEAP is responsible for providing hardware-abstracting APIs so that applications can access the hardware-components. To allow a communication between these separate process-spaces, inter-process-communication is needed.

LEAP uses an IPC specifically developed for it, who's usage is completely transparent for the application processes. This means that for the applications it does not make any difference whether they are linked with the service-libraries or with the IPC libraries.

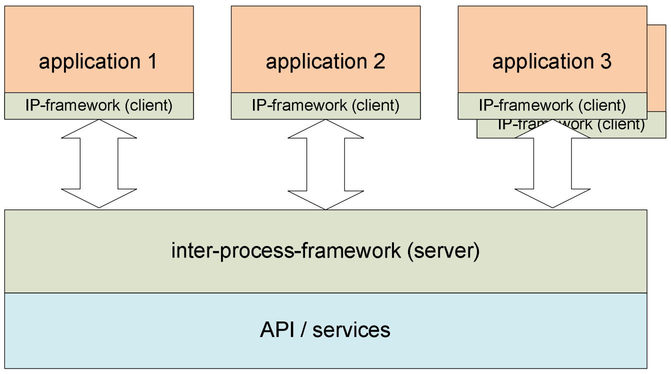
Overview

The requirements for the LEAP-IPC were the following:

- C++-APIs with polymorphism, multi-inheritance, STL and smart-pointers
- more than one application, 1:n-setup
- high API-object-complexity
- fast API-availability for the application-development-team with minimal error-injection during development
- ease of integration from the application-side, IPC-transparency
- efficiency in terms of data-flow
- high performance and small latency
- extensibility
- thread-safety

Other solutions already available in the market did not meet all of these requirements, which resulted in the decision to develop an original IPC-framework.

The LEAP-IPC is based on a client/server approach for decoupling the real implementation of the hardware-abstraction-functionality that resides in the service-process-space, and the application-process-spaces that contain functionality and use-cases. The IPC-server is responsible for handling creation and deletion of service-threads, communication-channels and with assigning the correct communication-channel to the right application-thread, all of these in a thread-safe context. The IPC is responsible for marshaling and unmarshaling the complex-objects.



Later changes in the API or introduction of a new API creates implementation efforts. Unlike other IPC-solutions that generate stubs, the LEAP-IPC reduces these efforts by following a code-generation-approach that completely generates all necessary code with only the input of the API-headers, resulting in a rebuild-reaction-time for the new API to be available to the application.

Communication between the different processes is done via TCP/IP-sockets, POSIX-pipes or shared-memory. Communication via shared-memory is the fastest but the least stable since faulty code could overwrite memory-areas. Communication via TCP/IP-sockets is the slowest, but it grants the possibility to perform IPC not only within one device but even remotely, e.g. from a PC to the embedded device for easy application-development.

For each thread in the service-process that communicates with an application-process or vice versa, a corresponding shadow-thread in the other process-space is created and a 1:1-communication-channel is established. This reduces the communication-complexity and enables a system to implement IPC with low latency and priority-support.

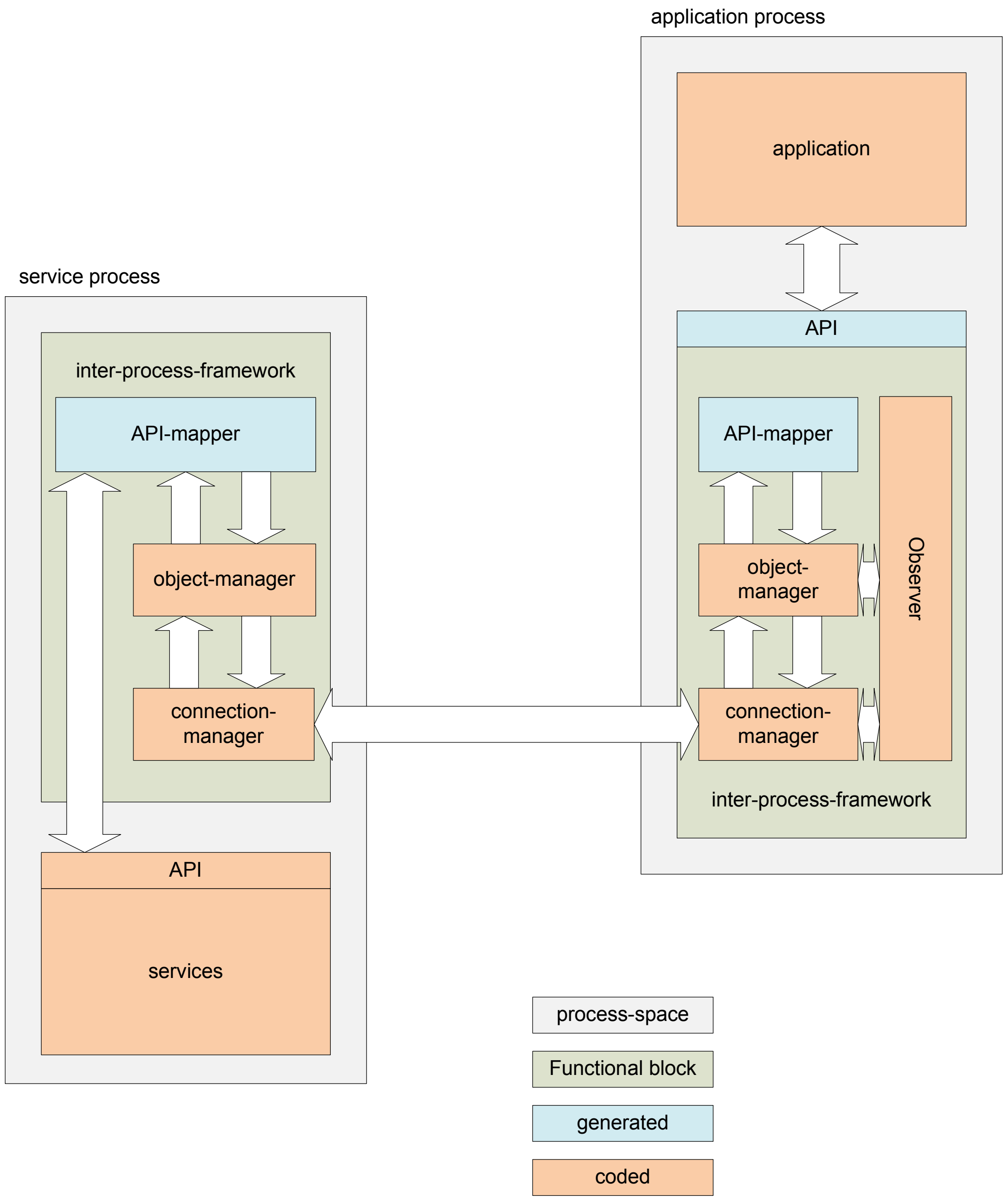
For each functional object in the service-process that is accessed by an application-process or vice versa, a corresponding shadow-object in the other process-space is created which only passes function calls to its real-object.

All these parts come together to create a completely transparent IPC-layer which can be used by the application-processes.

Layers

The LEAP-IPC contains the following layers:

- Connection-manager: transmission of data through TCP/IP-sockets, POSIX-pipes or shared-memory
- Object-manager: management of remote-objects being referenced by shadow-objects, mirroring the real-object in another process
- API-mapper: automatically generated source-code, implementing shadow-objects specific to a given API-interface that is parsed automatically for implicit forwarding of method-calls with arguments and return-values to enable absolutely transparent method-calls as if one owned the remote-object directly
- Observer: observation of application-processes, triggering tidy-up actions in the service-process when application-processes have died



The first layer, the connection-manager-layer, contains the management of all TCP/IP-sockets, POSIX-pipes or shared-memory being used to transport data between the service-process and application-processes. When a thread in an application-process calls a method of a shadow-object that mirrors an object in the service-process for the first time, a socket-connection is established to exchange administration-data, including which communication-channel to use, for further (faster) communication of method-calls and results in either direction from application to service-process or vice versa, i.e. for delivering events. A new thread in the service-process is created that is tied directly to the thread in the application that called the method in the shadow-object. This thread stays alive as long as the corresponding thread in the application does and handles only method-calls from this single thread. This avoids bottle-necks in the forwarding of method calls. So, from a logical point of view, a fix virtual 1:1-channel is created for every application-thread that calls API-methods. In the other direction, a channel is also created for every thread in the service-process and destination-application-process to which events are delivered by calling regular methods of specified consumer-objects or listener-objects which are physically placed in an application-process.

In each process, the second layer, the object-manager-layer, manages all objects that are passed through the IPC to another process or are received from it. When an object is transmitted through the IPC for the first time, a corresponding shadow-object in the receiving process is created and stored in a map. In following method calls, these objects are recognized implicitly as identical objects, also when referenced in different method-calls and possibly also in a different direction (i.e. sending instead of receiving). They are even identified as the same object when they are passed as pointers with the type of any super class in the case of inheritance. Objects can be passed as pointers or references or as shared-pointers of the Boost-library. When a shared-pointer (either with an object found in its own process or with a shadow-object referencing an object of another process) is transmitted to another process, a copy of this shared-pointer is kept implicitly by the IPC in order to avoid that the object is destroyed when all other shared-pointers pointing to the same object have disappeared in the sending process. Only when the received copy in the other process has been additionally removed, is this fact sent to the process that originally sent it, in order to remove the copy internally held by the IPC. This enables transparent usage of shared-pointers in method calls and as return-values as if they were only existing in their own process without anything that the application has to keep track of. Using shared-pointers is not forced. Also, raw-pointers can be passed through the IPC, but with the danger, as with any raw-pointer, that one process can destroy it although the other process still wants to use it or that the memory is not freed anymore if a process dies. Therefore usage of shared-pointers is recommended to increase stability because of transparency reasons. A direct deletion of a shadow-object is always followed by the deletion of the corresponding real-object in the other process and vice versa.

The third layer, the API-mapper-layer, consists of automatically generated code depending on the specific signature of the specified interface classes in the API. All class-headers in specified interface-directories are scanned and separated into so-called reference-classes and parameter-classes. A reference-class consists only of methods, either static or virtual, and never has variables. A parameter-class consists only of variables and never has methods. They can therefore be kept apart automatically and handled differently without the need of further error-prone manual assignment. For each reference-class, source-code for a corresponding shadow-class is created with the same interface, i.e. method-signatures. Furthermore, source-code is generated to receive marshaled method-calls, eventually grabbing the correct corresponding-object of which the method has to be called (if not a static one) and also the references (raw-pointers or smart-pointers) to already known shadow-objects or real-objects which are used as arguments and then passing it to the method of the real-object in the receiving process with the matching references as arguments. Arguments can also contain parameter-objects that contain only variables and no methods, or simple data such as numbers or strings. This data is just copied and passed to the corresponding method.

The fourth and final layer, the observer-layer, keeps track of the threads of the application-processes and the processes themselves which are still alive. When the death of an application thread is recognized, this information is passed to the connection-manager-layer which removes the data structures related to the logical channel that has been established to perform method calls from that application thread. When a process (not only a thread) has died, all shared-pointers that have been sent to the specified application process and are not yet deleted there (for example because the application-process may have been killed hard or terminated by error) are removed from the data-structures, too. Similarly, shadow-objects which have been transmitted as raw-pointers corresponding to an object in the dead application-process are removed then as well.

The IPC-module also offers other modules in the service-process the information when an application-process has died. This enables, for example, the management of resources that can only be used in one process at a time. So, when a process has died, the related resources can be released implicitly instead of being blocked forever.

Current status

The LEAP-IPC is fully implemented and is already in the field in several products as part of the LEAP-platform.

The following measurements were performed on an ARM 9 with 330 MHz. The method-parameters were two integer-values with a return-integer-value.

Communication channel	**method-calls/s**
TCP/IP-sockets	ca. 320
POSIX-pipes	ca. 570
shared-memory	ca. 1820

Conclusion

Altogether, our IPC module is a highly dynamic and completely transparent solution for forwarding method-calls from one process to another without needing to deal with low level technical aspects like opening connections and marshalling objects and other data. The number of application-processes and threads is completely flexible as well as the number of classes and instances of each class. Therefore, the benefit of a clear and comfortable interface such as available when working with only one process, can be combined with the safety resulting from the fact that API-services and applications run in different processes.