GENIVI®

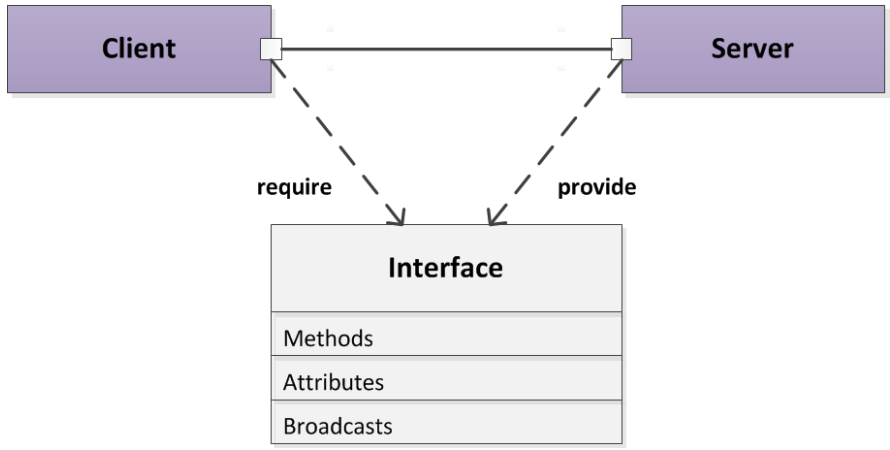
AMM OPEN DAYS

Common API C: Introduction

2015-10-21

Pavel Konopelko
Software Architect
Visteon

Purpose of Common API



- Client and Server communicate via the Interface that the Server provides and the Client requires
- Interface is defined in Franca IDL and include methods, attributes and broadcasts
- Interface can have multiple instances that are identified by their names
- Interface representation in programming language is pre-defined and is generated automatically together with the communication backend code
- Client and Server logic is backend-independent and compatible with any supported backend

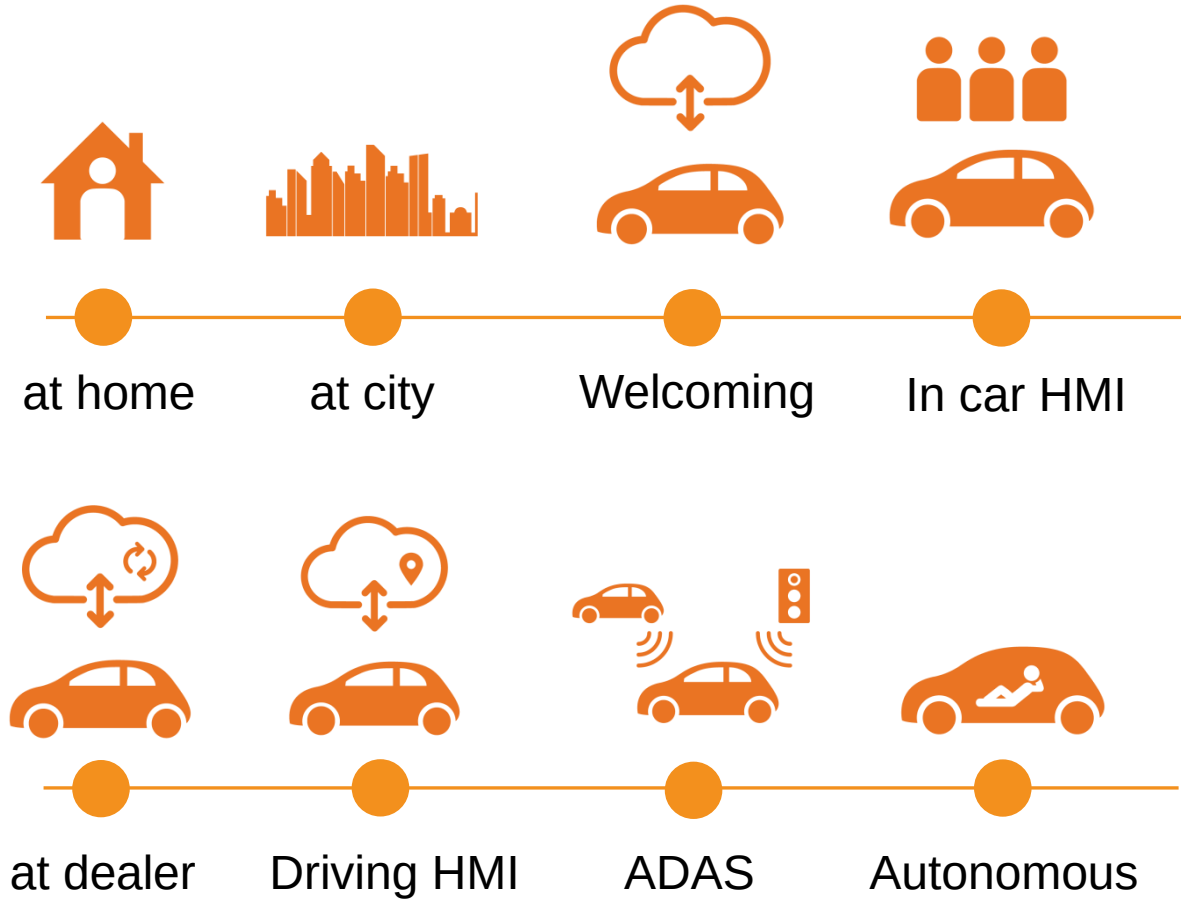
Related Projects

Common API C is related to other projects supported by GENIVI:

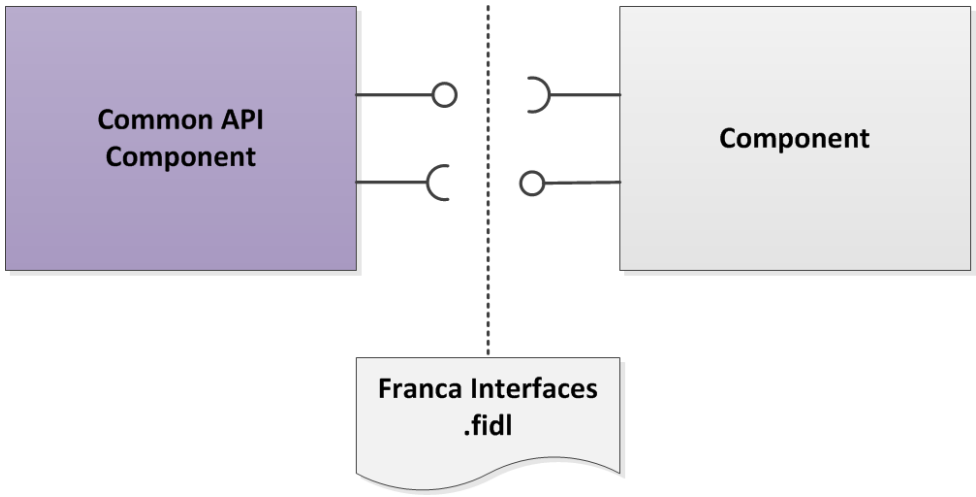
- Franca provides common IDL and infrastructure for code generation
  - http://franca.github.io/franca/
- Common API C++ implements C++ bindings
  - http://projects.genivi.org/commonapi
- Yamaica supports transformations between Franca IDL and UML
  - http://projects.genivi.org/yamaica/

Example of Common API Usage

- Visteon uses Common API and related technologies in the research work on adaptable software frameworks
- The goal is for the software to adapt to different drivers, to different passengers and to varying hardware devices
- More details at CES 2016

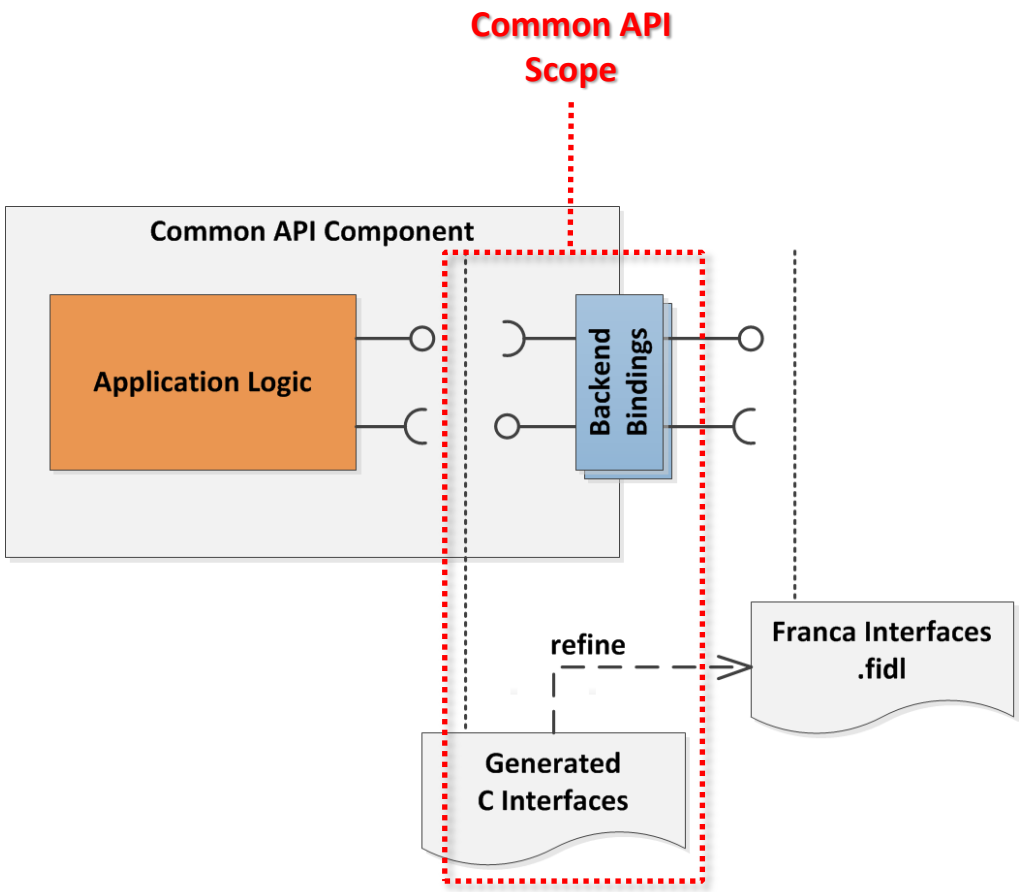


Exterior Component View



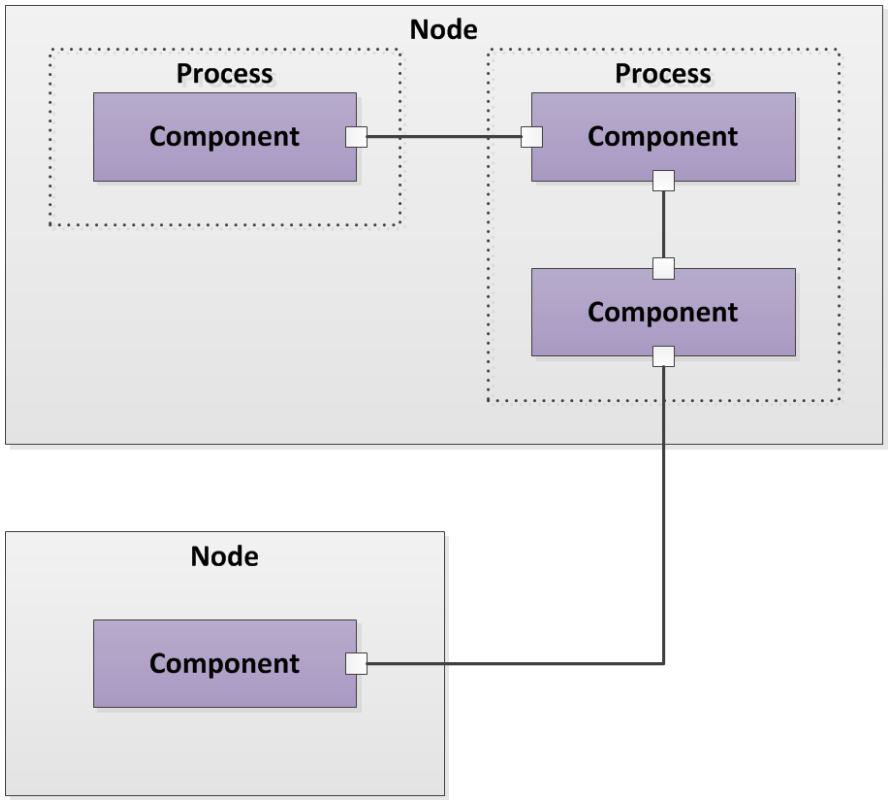
- Both client and server component implementations can be substituted by another one that implements a compatible Franca interface
  - This also includes components that do not use Common API, but implement the interface using a specific communication mechanism (e.g., D-Bus or SOME/IP) that has a defined transformation to Franca IDL
  - Compatible interface for servers is either the same or specialized; for clients it is either the same or generalized

Interior Component View



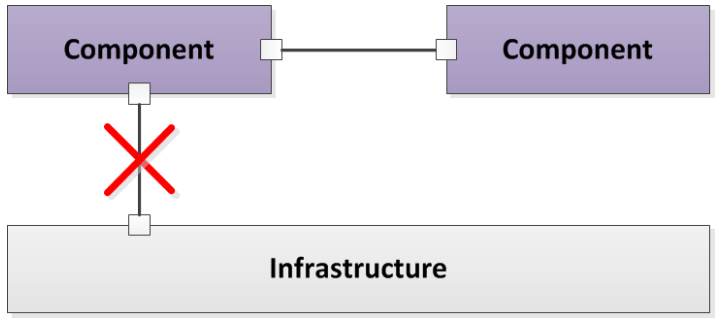
- Encapsulation
  - Backend Bindings encapsulate the knowledge of a particular communication mechanism (e.g., D-Bus, SOME/IP)
- Substitutability
  - Different backend implementations can be substituted for the bindings that are implemented with Common API
- Taken together, this insulates the Application Logic from the dependencies on a particular communication mechanism

Intended Usage: Flexible Deployment



- Deployment of the interacting software components must be flexible to support interaction within the same process, across different processes, and across different nodes
- In general, communication is asynchronous, data is passed by copy
- Interface design must take this into account (e.g., avoid passing large data chunks or very frequent interactions)

Intended Usage: Peer-to-Peer Interaction

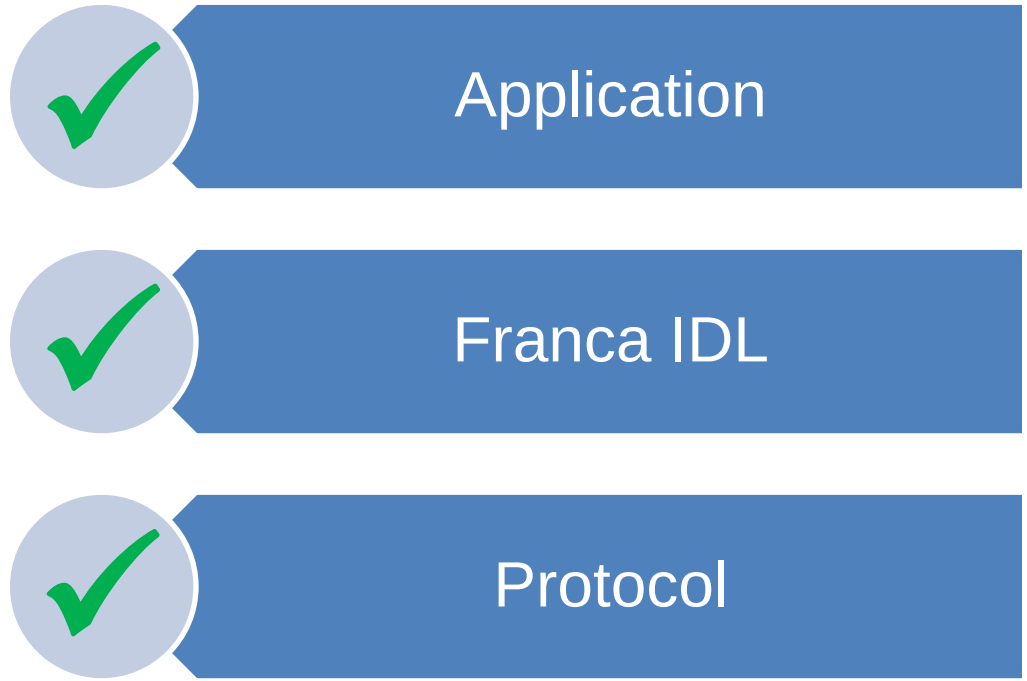


- Peer-to-peer (as contrasted to application-to-infrastructure) communication is the primary focus of Common API
- It does not attempt to provide a universal interface abstraction
- Functionality provided by the software platform infrastructure (e.g., device control, memory management, character data manipulation, etc.) has requirements that are not explicitly addressed by Common API

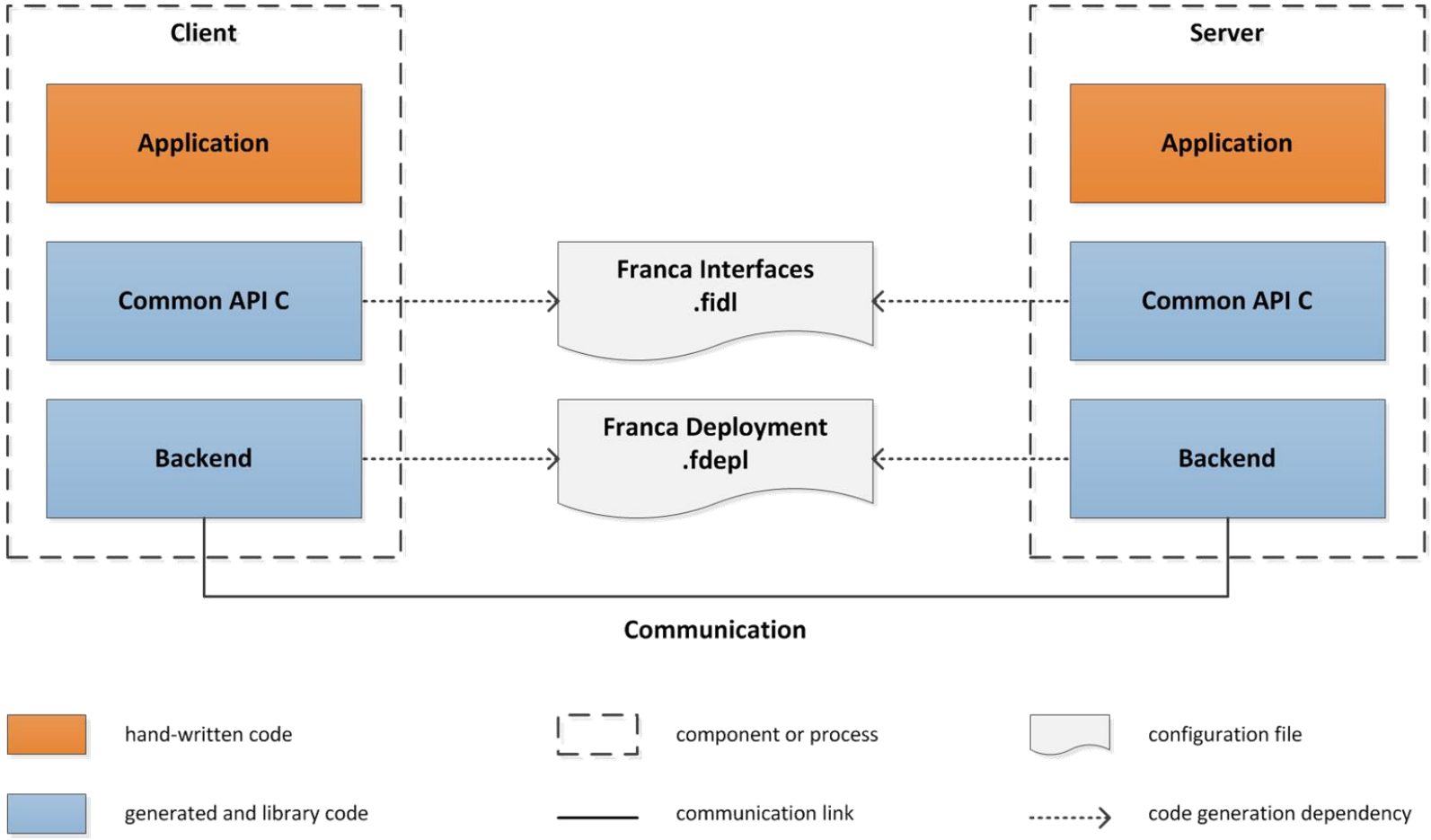
Interoperability Levels

Common API enables interoperability at three different levels:

- Application software modules
  - interaction via high-level abstraction of communication interfaces
- Interfaces defined in Franca IDL
  - interaction across programming languages and modeling tools
- Communication protocols
  - Interaction with non-Common API components



High-Level Architecture



Example of Application Code

```
/* ---- Calculator.fidl ---- */
package org.test
interface Calculator {
  version { major 0 minor 1 }
  method add {
    in { Double left
         Double right }
    out { Double sum } }
}
/* ---- client.c ---- */
cc_backend_startup();
cc_client_Calculator_new(
  "org.test.Server:/calculator:org.test.Calculator",
  NULL, &calculator);
cc_Calculator_add(calculator, 3.1415, 2.7182, &sum);
calculator = cc_client_Calculator_free(calculator);
cc_backend_shutdown();
```

```
/* ---- server.c ---- */
static int Calculator_impl_add(
    struct cc_server_Calculator *instance,
    double left, double right, double *sum) {
  *sum = left + right;
  return 0;
}
static struct cc_server_Calculator_impl impl =
{ .add = &Calculator_impl_add };
/* ... */
cc_backend_startup();
cc_server_Calculator_new(
  "org.test.Server:/calculator:org.test.Calculator",
  &impl, NULL, &calculator);
/* run backend event loop */
calculator = cc_server_Calculator_free(calculator);
cc_backend_shutdown();
```

Project Principles and Constraints (1/2)

- Align as much as possible with the Franca IDL mapping (e.g., for the data types) and implementation features (e.g., the approach to concurrency) implemented by Common API C++.
- Rely on the existing Franca framework for model transformations and code generation under Eclipse.
- Leave with applications the design choices related to concurrency (i.e., the main event loop vs. threading), to memory management (i.e., dynamic vs. static allocation) and to other major areas.

Project Principles and Constraints (2/2)

- Prioritize D-Bus/kdbus and in-process communication over other mechanisms for Linux environments.
- Support non-Linux environments and especially embedded, resource-constrained systems (e.g., do not require using dynamically allocated memory).
- Long-term, minimize the redundancy with the Common API C++ in the areas of Eclipse tooling and run-time support (e.g., backend libraries).

Current Project Status

- Project is run under the governance of GENIVI System Infrastructure EG
  - The project relies on the public GENIVI infrastructure (git, e-mail, wiki and bug tracker)
  - Compliance roadmap targets SC-P2 initially and SC-P1 once sufficiently mature
- Proof of Concept (PoC) is currently under development
  - The goal is to better understand the requirements and solution architecture
  - The PoC code is licensed under MPLv2; v0.1 released in August 2015
  - The PoC scope includes both the run-time libraries and the code generator
- Requirements and design ideas are documented in parallel to the PoC
  - See https://genivi-oss.atlassian.net/wiki/display/PROJ/Common+API+C+-+PoC (this will soon move to genivi.org)
  - The results are reviewed and discussed in the EG to agree on the project target

Features And Roadmap

Franca Features

- inheritance
- composite data types
- deployment specifications
- attribute access and update notifications
- broadcast sending and reception
- method invocation

D-Bus / kdbus backend -
multiple backends chosen at run-time -
in-process backend -
SOME/IP backend -

Backends

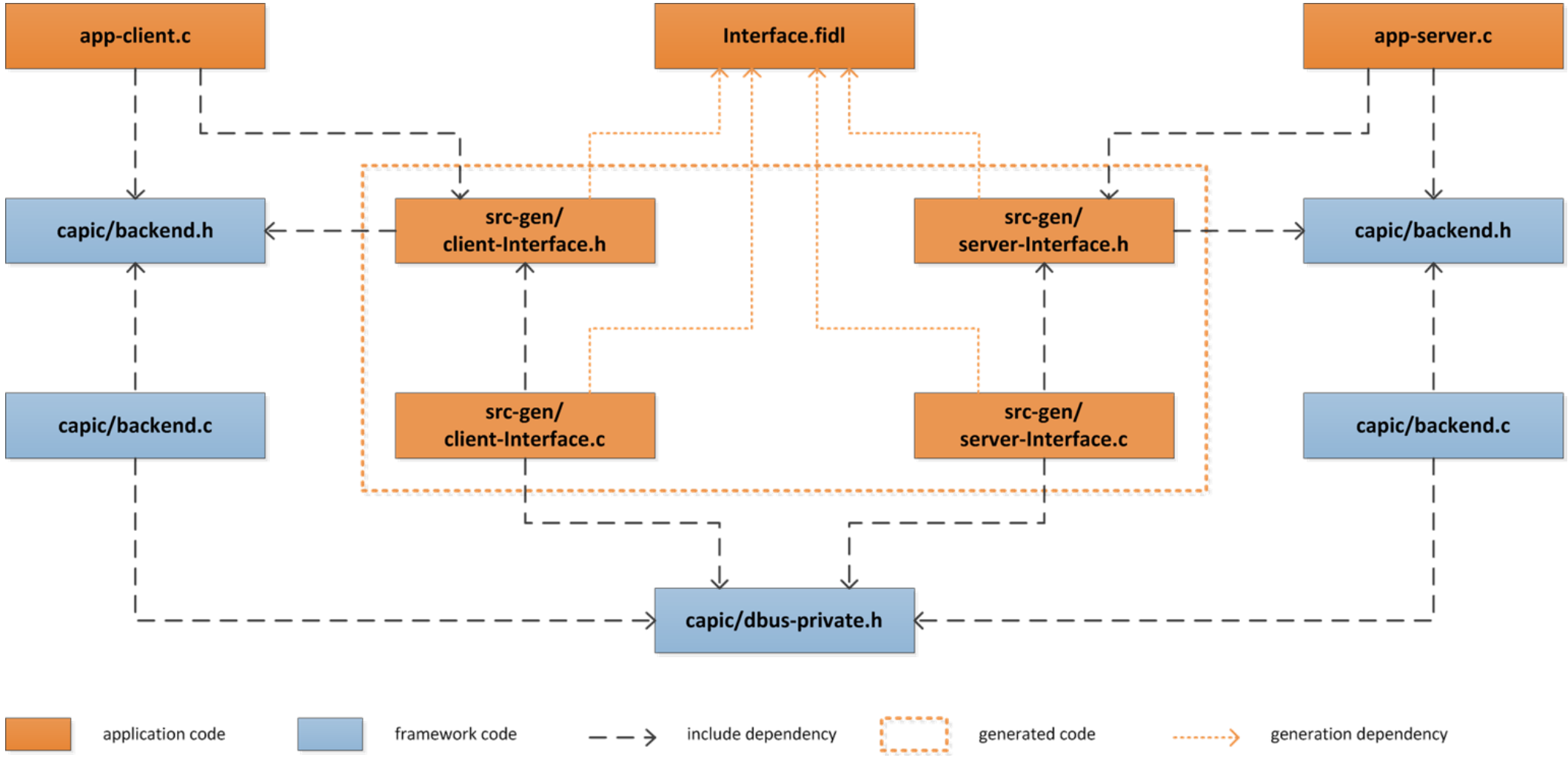
- Linux with main event loop
- application-owned memory management
- RTOS with tasks
- safety-critical requirements

Environments

PoC Development Progress

- Development approach is based on iterations
  - Manually develop application that uses certain Common API features (see below)
  - Incrementally implement corresponding aspects of the Common API C design
  - Extract shared implementation into a library and use the rest as generator test cases
- Functionality increments
  - Multiple interface instances and multiple interface implementations (DONE)
  - Asynchronous method invocations and backend event loop embedding (DONE, v0.1)
  - Code generation for currently supported features (DONE, v0.2 is due soon)
  - Support for Franca signals and attributes
  - Backend for in-process communication
  - Memory allocation managed by the application
  - Full support for Franca type system

PoC Status: Module Structure



PoC Status: Reference Code

- ‘Simple’ example
  - Server hosts two instances of the same interface with two different implementations
  - Client connects to both instances hosted by the Server
  - Client invokes a method synchronously (i.e. blocks until server responds)
  - Server responds to the method invocation synchronously (i.e. in the message handler)
- ‘Game’ example
  - Server hosts one interface instance that implements a state machine
  - Client implements another state machine that connects to the Server instance
  - Client uses GLib mail loop to invoke methods asynchronously (i.e., to receive a callback on server response)
  - Server uses sd-event to respond to method invocations asynchronously (i.e., the message handler defers the processing and response to a different handler)

PoC Status: Client-Side Methods

```
/* ---- Interface.fidl ---- */
package org.test
interface Interface {
  version { major 0 minor 1 }
  method methodName {
    in { InArg inArg }
    out { OutArg outArg } }
}
```

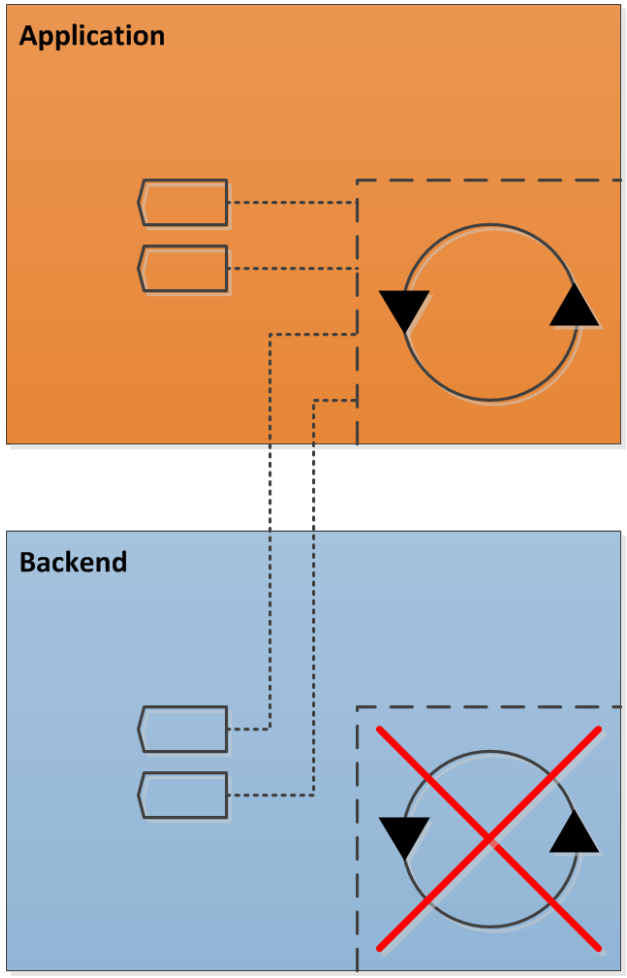
```
/* ---- src-gen/client-Interface.h ---- */
struct cc_client_Interface;
typedef void (*cc_Interface_methodName_reply_t)(
  struct cc_client_Interface *, OutArg);

int cc_Interface_methodName(
  struct cc_client_Interface *instance, InArg inArg,
  OutArg *outArg);
int cc_Interface_methodName_async(
  struct cc_client_Interface *instance, InArg inArg,
  cc_Interface_methodName_reply_t callback);

int cc_client_Interface_new(
  const char *address, void *data,
  struct cc_client_Interface **instance);
struct cc_client_Interface *cc_client_Interface_free(
  struct cc_client_Interface *instance);
```

PoC Status: Server-Side Methods

```
/* ---- Interface.fidl ---- */
package org.test
interface Interface {
  version { major 0 minor 1 }
  method methodName {
    in { InArg inArg }
    out { OutArg outArg } }
}
```

```
/* ---- src-gen/server-Interface.h ---- */
struct cc_server_Interface;
typedef int (*cc_Interface_methodName_t)(
  struct cc_server_Interface *, InArg, OutArg *);
struct cc_server_Interface_impl {
  cc_Interface_methodName_t methodName;
};

static int Interface_impl_methodName(
  struct cc_server_Interface *instance, ArgIn argIn,
  ArgOut *argOut);

int cc_server_Interface_new(
  const char *address,
  struct cc_server_Interface_impl *impl, void *data,
  struct cc_server_Interface **instance);
struct cc_server_Interface *cc_server_Interface_free(
  struct cc_server_Interface *instance);
```

PoC Status: Backend Event Loop

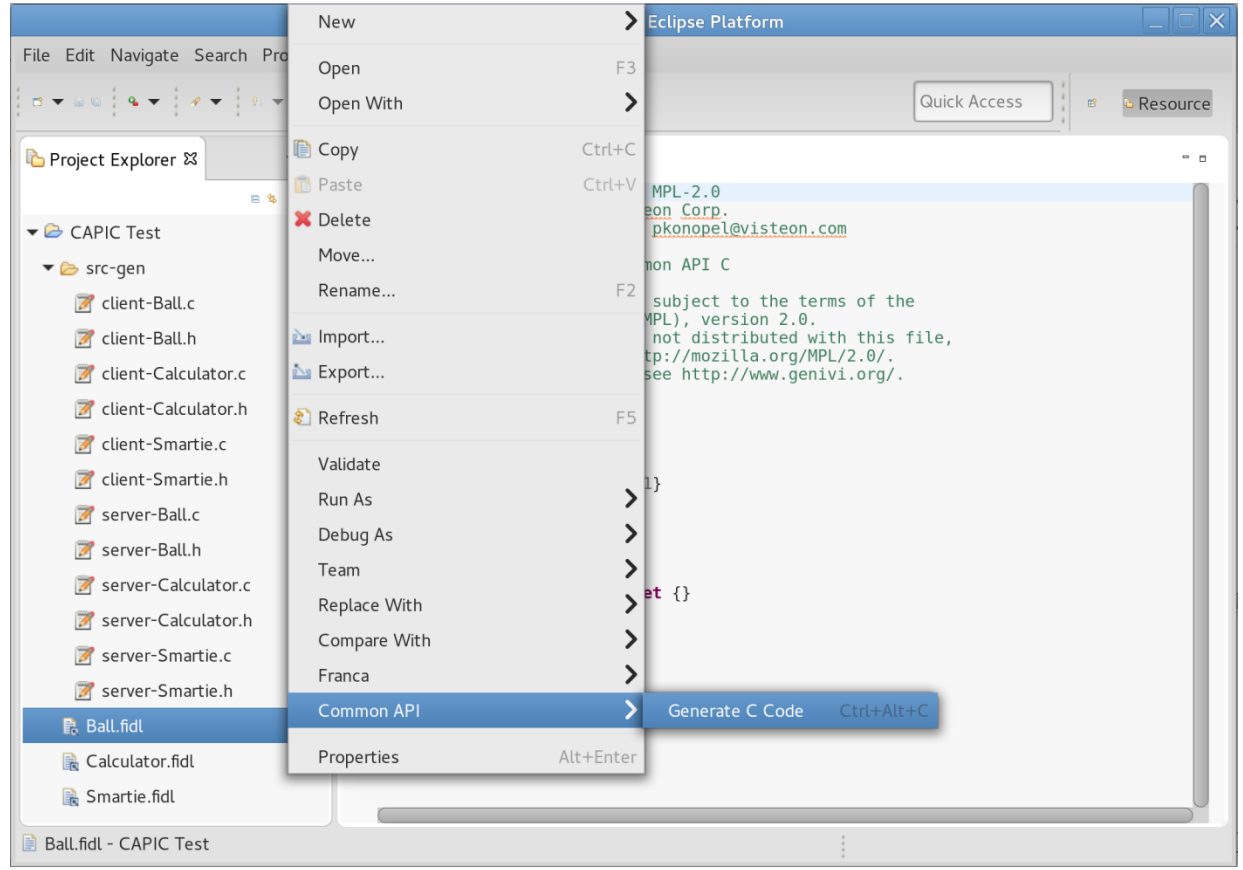



```
#include <capic/backend.h>

struct cc_event_context;

int cc_backend_get_event_context(
    struct cc_event_context **context);
void *cc_event_get_native(
    struct cc_event_context *context);
int cc_event_get_fd(
    struct cc_event_context *context);
int cc_event_prepare(
    struct cc_event_context *context);
int cc_event_check(
    struct cc_event_context *context);
int cc_event_dispatch(
    struct cc_event_context *context);
```

PoC Status: Eclipse Code Generator

Code generation for Common API C is supported via Eclipse UI plugin:

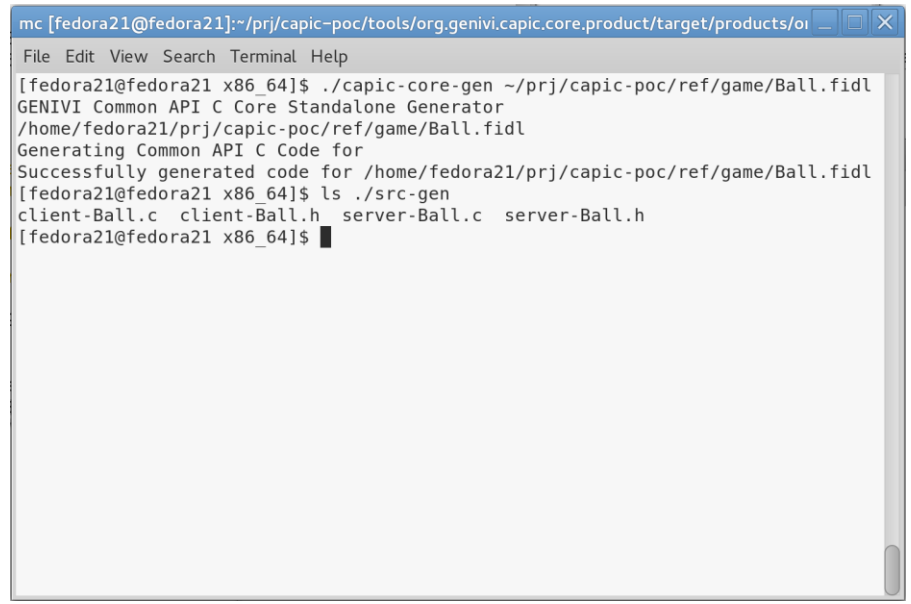
- Built with Java 1.7, Eclipse Mars SR1, Franca 0.10
- Command for .fidl files in context menu and keyboard shortcut
- Generates client and server code in the folder ‘src-gen/’ of the current project



PoC Status: Standalone Code Generator

Code generation for Common API C is supported via a standalone binary:

- Built with Java 1.7, Eclipse Mars SR1, Franca 0.10 (but only JRE is required for execution)
- Command line interface; .fidl file is specified by its absolute path
- Generates client and server code in ‘src-gen/’ under the current working directory



Project References

- Source code repository
  - http://git.projects.genivi.org/?p=common-api/c-poc.git;a=summary
- Mailing list
  - genivi-ipc@lists.genivi.org
- Wiki home page
  - https://genivi-oss.atlassian.net/wiki/display/PROJ/Common+API+C
- Bug tracker
  - https://genivi-oss.atlassian.net/projects/CC/issues/?filter=allopenissues

- *Beware: Wiki and bug tracker will soon move to* genivi.org

Thank You!

Questions?