Using an Embedded Data Server to Separate Concerns

Jakob.Wanner@sourceengineers.com

www.sourceengineers.com
Source Engineers GmbH
What do we offer

• Requirements Engineering
  • Hard- and Software Specification Documents
• System Architecture
• Continuous Integration
• Software project management
• Software Development
  • Implementation of complete software projects or Contract Work
  • Embedded Software for Microcontrollers and Application Processors
    • C/C++
    • RTOS, Linux or Windows CE
    • Qt GUIs
  • (Embedded) Web Interfaces
  • IoT and Connectivity
Challenges for Complex Software Projects

• Communication overhead
  • Number of communication channels increases with team size
    • (C = # communication channels, n = # stakeholders): C = n*(n - 1) / 2
    • E.g. n = 10 => C = 45, n = 20 => C = 190, n = 30 => C = 435 etc.

• Interface explosion
  • Increasing number of (class) interfaces created by the developers
  • Difficult to know which ones to use to solve a problem
  • When using many interfaces the possibility being affected by changes increases
  • Many dependencies
  • Problem complexity increases similarly like the communication overhead

➤ Solution: Compartmentalisation
  • Software is broken into components/modules
  • Decreases the number of public interfaces
  • Public interfaces are designed to be (more) stable
  • But…
The Monolith

... naive compartmentalisation often leads to an unorganised monolithic application
The Monolith

Why is this challenging?

• Complexity still increases exponentially
• Every component depends on many others
• Circular references are easily introduced accidentally
• Changes may impact many other components
  → Only most important changes are made because of the fear that something unexpected fails
  → Fragility increases
The Monolith

First attempt for improvement: Use software design principles

• Software Layers

• Dependency Inversion Principle: «High-level modules should not depend on low-level modules. Both should depend on abstractions»*

• Single Responsibility Principle: One class should only do one thing

* Robert C. Martin – Agile Software Development
The Monolith

Layered Design Approach

Oops, a cyclic dependency was introduced...
The Monolith

• Ensuring good practices for the whole project is a challenging task
• The project must afford a repository «gatekeeper» that reviews pull requests, supervises the architecture, consults team members and verifies that no unwanted dependencies are introduced
• Much work for an experienced developer...
• ...who then cannot contribute much own code anymore

➡️ Usually only big projects can afford that

...and in a complex project it’s still easy to accidentally break the architecture, e.g. by introducing circular dependencies
The Monolith

What happens when components are maintained or new components shall be added?

• Dependency on many other component’s interfaces
• Those interfaces may change over time
• Many people were involved in creating those interfaces
• To know which interfaces to use is still challenging
  → High communication overhead between the developers
Microservices

In larger systems to solve many of the problems that come with a monolithic application more and more often microservices are introduced

Pro: 
• One application/service per responsibility (Single Responsibility Principle/SRP)
• The code repository of a microservice is small and can be overseen and understood much easier
• A microservice is more maintainable by a small team
• The internals are hidden from external (mis-)use
• Different programming languages, tools, architectures allowed per microservice or a shared framework defines the basic inter-service communication
• Enforces a well-documented, expressive and stable interface because you don’t want to break other services
• Makes the application «self-repairable» in best case when certain services fail.
Microservices

Cons

• Might make the architecture and inter-service communication more complex

• As every service has its own API understanding a lot of APIs is essential for a new service

• Microservices often cross device boundaries and even if not, have a communication overhead (e.g. TCP/IP)

• Failure detection is more complicated (what if a service is unavailable because of a bad network connection?)
Microservices

Image: © Netflix
So microservices seem to be a good fit for large and complex systems....

But how can we scale down their architectural complexity for embedded systems?!?

• Preserve their architectural advantages
• Without introducing a lot of complexity
The Data Server stores, manages and provides data... via a unified interface which is common for all clients...

...which are completely separated on task/process level.
Data Server Approach

• Low complexity, easy to understand architecture also for new developers
• Data between tasks is only shared via the data server
• Complete separation of client concerns
• The unified interface guarantees low communication overhead and easy interfacing → as a task developer you only have to learn one interface
• Tasks can be developed independently of each other because there are no dependencies between them
• No shared memory between processes/tasks
• Lower supervising overhead → There can be separated code repositories for each task, so influencing other developer’s work accidentally is much more difficult
• Data Server and tasks can be implemented event-based and single-threaded which simplifies programming
• Works for RTOSes and ‘normal’ OSes
• Reusable for multiple projects
Data Server Pattern - Prerequisites

• Agreement on easy-to-use unified data interface
• Unified interface must fit all future purposes
• Well-suited if device borders are not crossed for the Server-Client communication

→ For very complex systems the Data Server Pattern might be an oversimplification
→ But it might also be a good complement within a complex architecture
Case Study: Embedded Data Server at myStromer AG

Images: © myStromer AG
Case Study: Starting Position

• New hardware revision of User Interface Module
• System-on-Chip with Linux instead of Microcontroller with RTOS
  • Expandability
    • Hardware (RAM, Flash, ...)
    • Software (language/compiler support)
  • Ecosystem
    • Allows use of standard Linux libraries, tools and drivers
• 500 MHz, 512 MB RAM and 512MB Flash
An application can be tested isolated from the rest of the system.
Case Study: Data Server Interface

The unified Data Server Interface is kept very simple:

- Key-Value Store Interface
- Applications can execute 4 operations on the parameters

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Param1Key</td>
<td>Param1Value</td>
</tr>
<tr>
<td>Param2Key</td>
<td>Param2Value</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Operations

- Set
- Get
- Subscribe
- Unsubscribe
Case Study: Data Server configuration

- The data server is configured via a JSON config file as a program argument
  - Running multiple data server instances in parallel is possible
- Data types: (u)int8/16/32/64, float, bool, string, byte array

```json
...  
socket: "/path/to/socket",
parameters: [
  {
    name: "param1",
    key: 1,
    type: "float",
    default: 0.0
  },
  {
    name: "param2",
    key: 2,
    type: "string",
    default: "Test"
  },
  ...
]
```
Case Study: Application Framework

App. Framework
Testing CLI

socket

Data Server

App. Framework
GUI

App. Framework
CAN Bus Communication

App. Framework
Control

App. Framework
...

source engineers
Case Study: Application Architecture

- Event-Loop-Based
  - Only active on file descriptor events, otherwise in blocked state
  - No CPU required when there is nothing to do

- Single-Threaded on application level
  - Simplifies application design (no semaphores, mutexes etc.)
  - No worries about multi-threading complications (deadlocks etc.)
  - Processor only had one core, so no performance gain would have been achieved by multi-threading anyway

- Reusable Application Development Framework
  - Shared between applications
  - Contains core features (Event Loop, Inter-Process Communication, ...)
  - Focus on software best practices (e.g. Dependency Injection)
  - Unit Tests

- Separation of object construction from object runtime
Case Study: Application Architecture

Data Server
- Inter-Process Communication Layer (Server)
- Business Logic Layer
- Data Layer
  - Cache
  - Persistence

Application Development Framework
- Event Loop
- Logging
- App Base Classes
- IPC
- Error Handling
- ...

Client Application
- Inter-Process Communication Layer (Client)
- Business Logic Layer

uses

uses
Case Study: Technology Stack

• C++14
• CMake (Build Tool)
• gcc (compiler)
• clang tools (tidy, sanitizer etc.)
• Yocto Linux
• Protocol Buffers (Data De-/Serialization)
• SQLite (Persistence Layer)
• Jenkins (Continuous Integration)
• Google Test Framework
What did Source Engineers do at myStromer?

• Requirements Engineering
  • Hardware Specification Document („Lastenheft“)
  • Hardware Supplier Evaluation

• Software Architecture
  • Main Software Architect
  • Data Server Approach
  • Architecture of Application Development Framework
  • Technology Evaluation

• Software Development
  • C++14 implementation of the Data Server and Framework
  • Testing CLI
  • Consulting
  • Qt Widgets GUI
Case Study: Results

- Complete rewrite of the former RTOS-based software written in C to a Linux-based software in C++14 plus many new features within < 1.5 years
- Version 1.0 was released on schedule!!!
- The strict separation of concerns into applications communicating via a simple data server interface eased development a lot
- Application development could be implemented independently from others
- Developers responsible for one application could hardly accidentally break code of others
- Reasoning about data flow, dependencies and interfaces was very easy
Questions?

www.sourceengineers.com