DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Research Internship Report

Checkpoint/Restore of IRQ sessions in Genode framework and L4 Fiasco microkernel

Author:	Harsha Sharma
Supervisor:	Prof. Dr. Uwe Baumgarten
Advisor:	Sebastian Eckl
Submission Date:	25th July 2018

I confirm that this research internship report is my own work and I have documented all sources and material used.

Munich, 25th July 2018 Harsha Sharma

Abstract

Development of new automotive technologies require many small microprocessors providing distinct functions which are also connected to each other. Integrating these small microprocessors on a single Electronic Control Unit (ECU) can significantly reduce cost and energy consumption. This comes with a major drawback of fault in one of the part of system affecting other parts. A real-time capable Checkpoint/Restore system can provide future mobility solution with fault tolerance and dynamic reconfiguration. This work is based on investigating and analyzing an existing C/R project called RTCR being developed at Technical University of Munich.

Contents

Abstract **iii**

1 Introduction **1**
1.1 Checkpoint/Restore Project 1
1.2 L4/Fiasco Microkernel vs Monolithic kernel 1
1.3 Genode 2

2 Background **3**
2.1 Genode OS Framework 3
2.2 Capability Delegation 3
2.3 Client and Server Relationship 4
2.4 Component Ownership 4
2.5 Services and Sessions 5
2.6 Shared Memory 6
2.7 Inter Process Communication (IPC) 6

3 Implementation **7**
3.1 Parent RTCR component 7
3.2 Checkpointing and Restoring Capabilities 8

4 Current Restrictions **9**
4.1 Missing IPC gate capability : 10
4.2 Missing IRQ capability : 10

5 Solution **11**
5.1 Structure of Target_Child 11
5.2 Custom IRQ session 12

6 Future work **14**

7 Bibliography **15**

1 Introduction

1.1 Checkpoint/Restore Project

Consider an automotive vehicle, where brake-control unit and car-navigation system are hosted by same computational platform, it is obvious that the navigation system is less critical that braking system and should never prevent normal functioning of car brakes. This leads to concept of mixed critical systems, which allow software components of different criticality levels to coexist on same hardware. A possible solution is to combine security and virtualization, providing abstractions for underlying hardware to each software component and hiding the software-hardware bindings. From the point of view of a software component, it is the only component running on hardware, while actual hardware chip in reality usually hosts multiple sub- systems.

If any of the components running on integrated hardware breaks or crashes, the RTCR project provides a transparent and real-time mechanism to checkpoint the state of an executing task, migrate the checkpointed state information to other ECU and restore the state on another ECU.

As soon as a failing node is detected, it is not considered reliable, making it difficult to checkpoint the state of task. So, the solution is to acquire the recent snapshot of the periodically checkpointed states and restore it. Once a checkpoint is transferred to target system, resources for the task to be restarted must be acquired. Then , the state contained in checkpoint is to be read and new process must be set up accordingly. When it is started up again, it can continue executing its task on different ECU.

1.2 L4/Fiasco Microkernel vs Monolithic kernel

Monolithic kernels are highly complex pieces of software which handle everything from managing resources (e.g. memory), accessing hardware and controlling user processes. This gives monolithic kernels , a privilege to control the whole machine. For example, a broken network driver can corrupt whole operating system. With virtual address spaces, monolithic kernel and hardware platforms provide mechanism to isolate concurrently running user processes. Microkernel-based systems use these techniques not only for user applications but also for device drivers, file systems, and

other typical kernel-level services.

Fiasco is a preemptible real-time kernel supporting hard priorities. It uses non-blocking synchronization for its kernel objects. This guarantees priority inheritance and makes sure that runnable high-priority processes never block waiting for lower-priority processes. It also provides flexibility with multiple virtualization options.

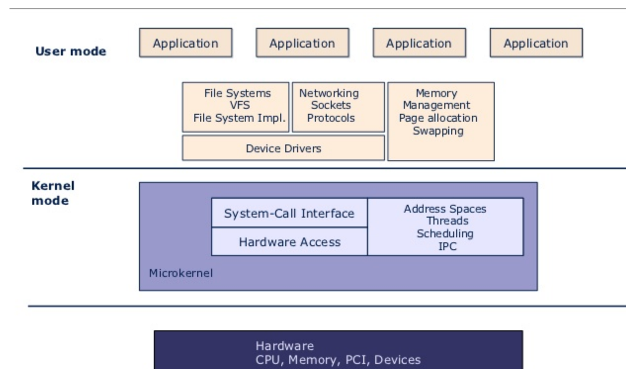

Fig.1 : Microkernel architecture

1.3 Genode

It is an operating system framework for building highly secure operating systems. It can be deployed on a variety of microkernels(NOVA, Fiasco.OC, L4/Fiasco). The system is based on a recursive structure. Each program is executed in a dedicated sandbox and gets granted only those access rights and resources that are required to fulfill its specific purpose. Programs can create and manage sub-sandboxes out of their own resources, thereby forming hierarchies where policies can be applied at each level.

The framework provides mechanisms to let programs communicate with each other and trade their resources, but only in strictly-defined manners. A Genode user-level component is unaware of the underlying kernel and thus most of characteristics of the Fiasco.OC L4/Genode.

2 Background

2.1 Genode OS Framework

Genode is a component based architecture where each component has a protection domain (PD) that provides an isolation execution environment. Genode introduces notion of RPC objects and capabilities associated with them. An RPC object provides a remote procedure call interface and has an associated capability. A capability refers to a specific RPC object and used to call functions of RPC object. During creation of RPC objects, capability in cap map in genode and an object identity is created which refers to capability in cap space in kernel. Object identity is the relation between cap map in genode and cap space in kernel. For each PD, kernel cap space is namespace local to PD, so kernel cap space can be accessed in a specific PD in genode. One component or one PD can have multiple RPC objects.

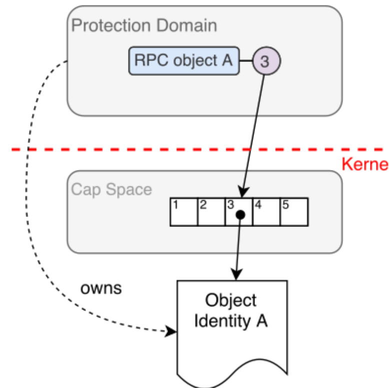

Fig.2 :Relationship between RPC object and its object identity.

2.2 Capability Delegation

RPC object in one protection domain can be delegated by kernel to have different capability in another PD, so that it can accessed by other PD's also. But capabilities

can be delegated only inside the protection domain in which it was created. Capability delegation does not hand over ownership of RPC objects. Only owner of RPC object can destroy it along with corresponding object identity and kernel space.

2.3 Client and Server Relationship

A component that uses RPC object is called client component and corresponding PD is called client PD. A component which owns RPC object is called server component. Client and server are different threads in their respective protection domains. Server thread is called entrypoint thread which becomes active only when call from a client enters the server PD or when asynchronous notification comes in. Entrypoint thread associate a capability with RPC object in genode and kernel space. Entrypoint thread, i.e. server component created RPC object with associated genode capability and object identity. One server can have many clients. The badge value , which is a system-globally unique object ID is used at the server side as a key for looking up the RPC object that belongs to an incoming RPC request.

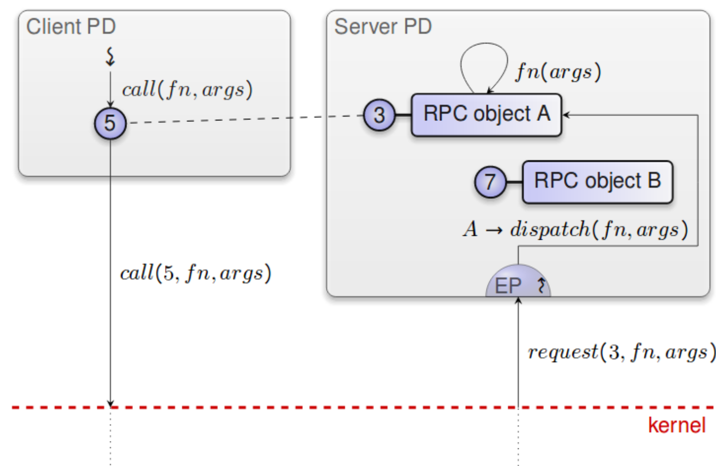

Fig.3: Client and Server Relationship

2.4 Component Ownership

Each component has a parent component which owns it. Each new component is created with empty PD. Parent populate PD with code and data and create a thread that executes code within protection domain. At creation time, parent installs parent

capability into new PD. The parent capability enables child to perform RPC calls to parent. Server and Parent components are not same, but both child component and server component will have a common parent which is init component in general cases.

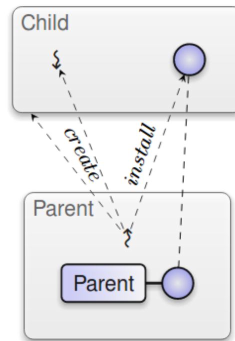

Fig.4: Parent and Child Relationship

2.5 Services and Sessions

Child needs to announce parent about services it provides. In order to provide a service, a component needs to create an RPC object implementing the so-called root interface. The root interface offers functions for creating and destroying sessions. Client can also create session by issuing session requests to parent. When the parent receives a session request from child, it can deny the service, provide the service or even direct session request to other child.

If the parent decides to provide the service, it hands out a session capability in reply to session request to child. Low level physical resources are represented as services provided by PD session component. Physical memory is handed out and accounted by PD service of core. At boot time, core component creates a PD session.This PD session is designated for init component, which is the first and last child of core component. Physical memory is made available as so called RAM dataspaces allocated from core's PD service. Similarly. CPU is represented by CPU service and interrupts are represented by IRQ service.

In order to access a resource, a component has to establish a session to the corresponding service. A service is implemented as root RPC object, which can create , upgrade and destroy sessions. A session RPC object is a service instance which provides service specific methods. Session RPc objects can be requested through the parent component and normal RPc objects are obtained from methods of session RPC object.

2.6 Shared Memory

Each dataspace is a distinct RPC object. Each component in possession of dataspace capability can make the dataspace component visible in its local address space. By means of delegating capabilities, components can establish shared memory.

2.7 Inter Process Communication (IPC)

At the lowest level, the kernel's IPC mechanism is used to transfer messages back and forth between client and server. The client side of the communication channel executes an IPC call operation with a destination capability, a send buffer, and a receive buffer as arguments. The send buffer contains the RPC function arguments, which can comprise plain data as well as capabilities. The IPC library transfers these arguments to the server via a platform-specific kernel operation and waits for the server's response. The response is returned to the caller as new content of the receive buffer. Building block of IPC in Fiasco is IPC gate kernel object, which is created by kernel factory object.

3 Implementation

Main purpose of RTCR is to monitor the state of child and periodically checkpoint its state onto a seperate memory location. Target child is the target process which needs to be checkpointed. The intercepting component offers these RPC objects for target process. The intercepting component also intercepts and monitor method invocation initiated by its interface user. Online storage is meant for intercepting component and stores live values of RPC objects. Offline storage is used for storing process data from a certain time (i.e. from last checkpoint). Checkpointer component processes data from online storage and store them in offline storage. Restorer component requires data from offline storage and access to create RPC objects.

3.1 Parent RTCR component

The idea of approach is, the parent component checkpoints and restores a child component , because it knows the resources which it provides to child at the creation time or when requested by child.Therefore, parent component provides custom session RPC object which intercept the usage of child component to real RPC objects. Instead of providing core's PD, CPU, and RAM session at the creation time of the child, the parent component provides its own PD, CPU, and RAM session which internally use core's services. Furthermore, whenever the child requests a session to an arbitrary service, the parent provides a custom session of its own which also uses the real service internally. This imposes a performance disadvantage during the runtime of the target by extending its execution time by doubling the waiting time for an RPC function call.

RTCR is the parent component and uses custom RPC objects which monitor access of the child to these objects. By monitoring the access, RTCR can recreate the internal state RPC objects by reissuing same calls as the child. By intercepting all session requests and invocation of session functions, the parent can restore state of RPC objects used by child. Restoring state involves creating RPC objects, injecting capabilities into cap space and restoring the state.

When the restoration mechanism starts, identify the RPC objects created by bootstrap mechanism and associate them with checkpointed state with checkpointed object data and update their state to checkpointed state and recreate the RPC objects which were not automatically created by bootstrap mechanism and update their state.

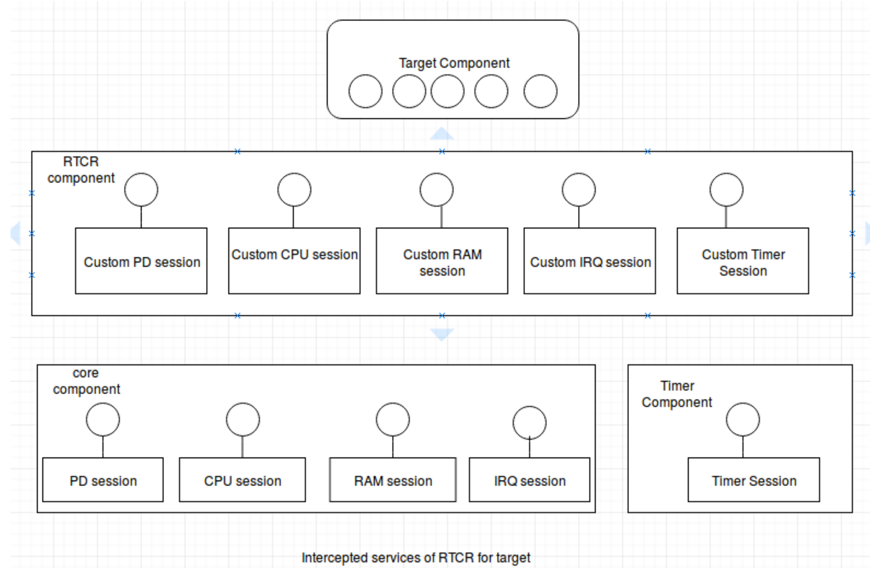

Fig. 5: Intercepted session of RTCR for target

3.2 Checkpointing and Restoring Capabilities

Capability of manually recreated RPC objects have to be inserted into the capability space. Reading the capability space of target component can be realized by reading the capability map stored in target's memory as capability space in kernel is a mirror of capability map in genode.

Capability map has to be readjusted to new badges of new RPC objects. To restore the capability space not only the kernel objects have to be restored, but also the locations in the capability space which point to them. Additionally, the capability map which resides in the address space of the target has to be adjusted. Each recreated RPC object will have a new badge, but the checkpointed memory content will still reference the old badges. Thus, the new badge has to be replaced with the corresponding old badge in the capability map.

4 Current Restrictions

Some of the capabilities in kernel cap space were missing, though all the capabilities in cap map in genode are checkpointed and restored correctly.

Before Checkpoint:

000000	–	00141-SW Task	–	–
000004	–	–	–	–
000008	0017a-SW Gate	00173-SW Gate	00176-SW Gate	00174-SW IRQ
00000c	00187-SW Gate	0018a-SW Gate	00188-SW IRQ	00195-SW Gate
000010	00198-SW Gate	00196-SW IRQ	–	–
000204	0016d-SW Gate	0015e-SW Gate	00157-SW Gate	00151-SW Gate
000208	00153-SW Gate	0017d-SW Gate	00180-SW Gate	00183-SW Gate
00020c	00178-SW Gate	00184-SW Gate	0018d-SW Gate	–
000210	–	–	–	0018f-SW Gate
000214	00191-SW Gate	00193-SW Gate	0019a-SW Gate	00144-SW Gate
000218	00143-SW IRQ	0019c-SW Gate	–	–
00021c	–	–	001a2-SW Gate	001a4-SW Gate
000220	001a5-SW Gate	–	–	–

After restore:

000000	–	001d8-SW Task	–	–
000004	–	–	–	–
000008	00212-SW Gate	0020b-SW Gate	0020e-SW Gate	0020c-SW IRQ
00000c	0021e-SW Gate	00221-SW Gate	0021f-SW IRQ	00226-SW Gate
000010	00229-SW Gate	00227-SW IRQ	–	–
000204	00205-SW Gate	001f6-SW Gate	001ef-SW Gate	001e8-SW Gate
000208	001eb-SW Gate	00215-SW Gate	00230-SW Gate	00217-SW Gate
00020c	00210-SW Gate	00218-SW Gate	00223-SW Gate	–
000210	–	–	–	–
000214	–	00219-SW Gate	0022c-SW Gate	001db-SW Gate
000218	–	–	–	–
00021c	–	–	00236-SW Gate	0021b-SW Gate
000220	0021a-SW Gate	–	–	–

As it can be seen, badges for capabilities for kernel objects after restoring process are updated. It can also be seen that capabilities for two IPC gates and a capability for IRQ kernel object is not restored correctly.

4.1 Missing IPC gate capability :

The child creates an Genode::Entrypoint object, it creates a capability to a thread kernel object by the same method call as is used to create capabilities for session RPC objects. This capability has to be identified and restored to allow the child to maintain its service on the migrated node.

4.2 Missing IRQ capability :

IRQ kernel object represents async information and kernel maps interrupt lines to IRQ kernel objects. IRQ kernel object is created using l4_factory_create_irq by the factory object. In genode, interrupts are handled by signal handler session which manages asynchronous notifications for inter process communication. In RTCR project, the incoming and outcoming signals are not restored,reason being, that when the target is restored, the other components which issued the communication messages are not there on the same node and outgoing signals are not necessary to checkpoint, because the target does not expect any answer. Also, in L4 Fiasco, the IRQ kernel object needs to be attached to IPC gate via l4_attach_irq call, when restoring the state of target component. So we need a custom IRQ session provided by parent interface which intercepts live session data in online storage. This information can be used while restoring the target state and IRQ kernel object can also be attached to corresponding IPC gate while restoring.

5 Solution

5.1 Structure of Target_Child

Target Child is the the target process which need to be checkpointed.

```
Target_Child::Target_Child(Genode::Env &env, Genode::Allocator &md_alloc,
Genode::Service_registry &parent_services, const char *name, Genode::size_t granularity):
    _in_bootstrap(true), //indicates if process was created during bootstrap
    _name(name), // name of target_child
    _env, // Genode env variable
    _md_alloc(md_alloc), // Allocated memory for process, e.g heap
    _resources_ep(_env, 16*1024, "resources_ep"), // resources entrypoint thread
    _child_ep(_env, 16*1024, "child_ep"), //child entrypoint thread
    _granularity(granularity), // used for incremental checkpointing
    _restorer(nullptr), //attribute represents restorer component
    _custom_services(_env, md_alloc, _resources_ep, _granularity, _in_bootstrap),
    //custom services provided by parent of target_child which refers to root interface.
    _resources(_env, _name.string(), _custom_services),
    // resources provided by parent to child component
    _initial_thread(_resources.cpu, _resource.pd.cap(), _name.string()),
    _address_space(_resources.pd_address_space()),
    _parent_services(parent_services),
    _child(nullptr)) // _child attribute represents child component and is a pointer to
    Genode::Child object.
```

Custom services provided by parent root interface of target-child :

```
Target_child::Custom_services
    pd_root: Pd_root*
    cpu_root: Cpu_root*
    ram_root: Ram_root*
    rm_root: Rm_root*
    log_root: Log_root*
```

```
    timer_root: Timer_root*
    irq_root: Irq_root*
```

Resources:

```
Target_Child::Resources

    pd:Pd_session_component
    cpu:Cpu_session_component
    ram:Ram_session_component
    rom:Rom_session_component
```

Target child creates root RPC objects for handling child's session requests. Root RPC objects create session RPC objects which create normal RPC objects.

Root RPC objects which provide the parent interface request core or specific component for session RPC objects internally and creates normal RPC object on session method invocation. State of root object is a list of created session RPc objects and in this way, it monitors the creation of session RPC objects. Session RPC objects is partly a list of created normal RPc objects and other attributes like signal context capability for signal reception. The method invocation of normal RPC objects which change state of target only needs to be checkpointed.

5.2 Custom IRQ session

Method invocation of IRQ RPC object which change its state are:

```
ack_irq() //Acknowledge handling of last interrupt - re-enables interrupt reception
sigh(Genode::signal_context_capability cap) //Register irq signal handler
info() //Request information about IRQ, e.g. on x86 request MSI address and MSI value
to be programmed to device specific PCI registers.
```

Irq_root class has a list of Irq_session_component which are requested by child. An Irq_session_component is created by _create_session_method and creation arguments are stored in _parent_state object. The _upgrade_session method extends the ram quota of Irq_session_component and also stores it in _parent_state. A Pd_session_component is closed by _destroy_session method which removes object from list of Pd_root.

Online state of irq session

```
struct Rtcr:: Irq_session_info : Session_rpc_info
{
    Irq_sesison_info(const char * creation_args, bool bootstrapped)
    :
        Session_rpc_info(creation_args, " ", bootstrapped)
    { }
};
```

Offline state of Irq session:

```
struct Rtcr::Stored_irq_session_info : Stored_session_info,
 Genode::List<Stored_irq_session_info>::Element
{
    Stored_irq_session_info(Irq_session_component &irq_session,
    Genode::addr_t targets_kcap)
    :
            Stored_session_info(irq_session.parent_state().creation_args.string(),
                                irq_session.parent_state().upgrade_args.string(),
                                targets_kcap,
                                irq_session.cap().local_name(),
                                irq_session.parent_state().bootstrapped)
        { }
};
```

When checkpointing from last state and restoring state in offline storage, there is no need to create objects everytime in offline storage comparing to last state. For each RPC object in online storage, a stored RPc object is searched in offline storage. If one is found, it is updated with new values. If no such object is found, a new stored RPC object is created and then updated. In L4 Fiasco, we also need to attach IRQ kernel object to corresponding IPC gate via l4_irq_attach call via restoring state of RPC object.

6 Future work

Currently, we create a child which creates a session request for IRQ session to parent component. The parent fails to bind the interrupt no to ICU and replies with an invalid session capability. We need a device driver which sends an interrupt signal which can be checkpointed by parent component in genode, to test he working properly. The kernel space capabilities before checkpointing and after restoring can be compared in similar way as done before.

7 Bibliography

1. S. Eckl, D. Krefft, and U. Baumgarten. "COFAT 2015 - KIA4SM - Cooperative Integration Architecture for Future Smart Mobility Solutions." In: Conference on Future Automotive Technology. 2015.

2. Denis Huber - Design and Development of real-time capable Checkpoint/Restore Mechanisms for L4 Fiasco.OC/Genode,Master's Thesis 2016

3. Extension of L4 Fiasco.OC and Genode OS Framework by the Concept of a Distributed Shared Memory, Bachelor's Thesis, 01/2016

4. Genode foundations 17.05 book