

Jari Operating System platform

December 2010

Contents

I	Platform architecture design	7
1	General architecture overview	9
1.1	Introduction	9
1.1.1	Jari OS platform general description	9
1.1.2	Platform purposes and targets	9
1.1.3	History	10
1.2	Generic platform parts	11
1.2.1	Microkernel	12
1.2.2	Domain holder	13
1.2.3	General libraries	14
1.2.4	RPCv7 and vnode abstraction	15
1.2.5	IDL	15
1.3	Platform layers	16
1.3.1	Driver model	16
1.3.2	File system layer	16
1.3.3	Terminals layer	16
1.3.4	Network layer	16
2	μString third-generation microkernel	17
2.1	Introduction	17
2.2	Memory management	17
2.2.1	Basics of operation	17
2.2.2	Memory objects	17
2.3	Interrupts, events and IPC	17
2.3.1	Interrupts	17
2.3.2	IPC	17
2.4	Scheduling	17
2.5	Access control	17
2.6	Syscalls	17
3	General abstractions	19
3.1	Introduction	19
3.2	IPC abstractions	19
3.3	Data access abstraction	19

3.4	Data structures	19
4	IDL	21
4.1	Introduction	21
4.1.1	Document version	21
4.2	Basic concepts	21
4.2.1	Protocol interfaces nesting	23
4.2.2	QoS	23
4.2.3	Distributed message processing	24
4.3	IDL	26
4.3.1	Interfaces	26
4.3.2	Types used in interface descriptions	26
4.3.3	Interface description	33
4.3.4	Creating a complete interface	36
4.4	Using the generated code	38
4.5	Backends	38
4.6	Compiler options	38
4.7	Examples	38
4.8	Build and install	38
4.9	Extend IDL	38
5	Domains subsystem	39
5.1	Introduction	39
5.1.1	Used terms	39
5.1.2	Use cases	40
5.1.3	General overview	40
5.1.4	Advantages and disadvantages	41
5.2	Implementation overview	42
5.2.1	Initialization	43
5.2.2	Microkernel-side support	44
5.2.3	User-space support	45
5.3	Using name spaces	45
5.3.1	Default root name space	45
5.3.2	Multispace system	45
5.3.3	Configuration	45
5.4	Benchmarks	45
6	Filesystem layer	47
7	Driver model	49
7.1	Overview	49
7.2	IPC classes	52
7.3	Device class protocols	52
7.4	Device interfaces	53
7.5	Device manager	53
7.6	Middle and low level device drivers	54

<i>CONTENTS</i>	5
7.7 Event model	55
7.8 Configuration parameters	55
8 Terminals layer	57
9 Network layer	59
II Platform implementation	61
10 Domain holder	63
10.1 Introduction	63
10.1.1 Preventing overload	64
10.1.2 Used data structures	64
10.2 Task and resources operational	66
10.2.1 General overview	66
10.2.2 Tasks and resources storage	67
10.2.3 Tasks hierarchy storage	68
10.3 Filesystem points operational	69
10.4 System bus operational	69
10.5 Domains	69
10.6 Techniques used to prevent DDoS attacks	69
10.7 Translators	69
11 Filesystem layer	71
11.1 Introduction	71
11.1.1 Network filesystem	71
11.1.2 Distributed filesystem	71
11.2 General vfs library	71
11.3 Remote procedures	71
11.4 Filesystem unions	71
11.5 Filesystem container library	71
11.6 Block resources data access library	71

Part I

Platform architecture design

Chapter 1

General architecture overview

1.1 Introduction

This book is intended to be a guide for those who wants to develop, use, research the Jari OS platform.

It will describe all the architecture from the basics to the deep of the components implementation.

It's not a final document, keep in my mind to let this book up to date.

1.1.1 Jari OS platform general description

Jari OS is not an Operating System itself, it's a platform for Operating Systems.

Platform based on the microkernel and multi-service architecture and follows minimality principles. Each additional component might be turned off or excluded from your OS to exceed your specific needs.

Platform can support several standards depends on your needs and domestic area.

Being the microkernel and multiservice Jari OS implements all objects like file, a set of objects (subsystem) like a special filesystem, this allows to have a one general interface to the objects.

Basically Jari OS supports POSIX standard and follows UNIX-principles, i.e. it's a UNIX-like system, however you may build other system on top of the platform.

1.1.2 Platform purposes and targets

Platform was developed as universal solution for industry and automatization area, the general decisions was made to conform industry standard and requirements.

Of course it doesn't means that the platform cannot be used for other solution, and/or be extended to support general purpose features.

1.1.3 History

Initially, Jari OS was developed like a new approach to design universal operating system with a good scalability and flexibility. To conform this target idea "everything is a file" was taken.

First development attempt has began in september of 2005 and continued several month, for the first design hybrid kernel with a set of the services was taken. During the first period of the development several ideas was tested. After those development stalled for the next several years.

In july of 2008 development goes continue, for the first stage only research was applied. During this research Jari OS has gone from the exokernel design architecture to the actual microkernel multi-service design.

Till december 2009 Jari OS was actually the operating system for the specific purposes. All development and research was going under commercial support.

When donations and support has ended Jari OS was oriented to create a platform - a complete designed system with many optional components to suit wide range of the domestic operating systems.

Since september 2010 research and development of the Jari OS supported by the Jari OS non-profit organization (Jari OS ry, Helsinki, Finland).

1.2 Generic platform parts

Jari OS platform has a three main parts:

- Microkernel (μ String - Jari OS third generation microkernel)
- Domain holder (Special service that serve all of higher level resources and abstractions)
- The set of the generic system libraries

This triplet is the core of the platform, it cannot be excluded. However, it's relatively small part of the platform, but it allow to implement other system features, support more and more standards, hardware, protocols and so on.

To communicate with the system platform provide two low level mechanisms: syscalls and IPC provided by the microkernel. Since microkernel supports only basic, low level operations, the most calls going throught the RPC calls to the services.

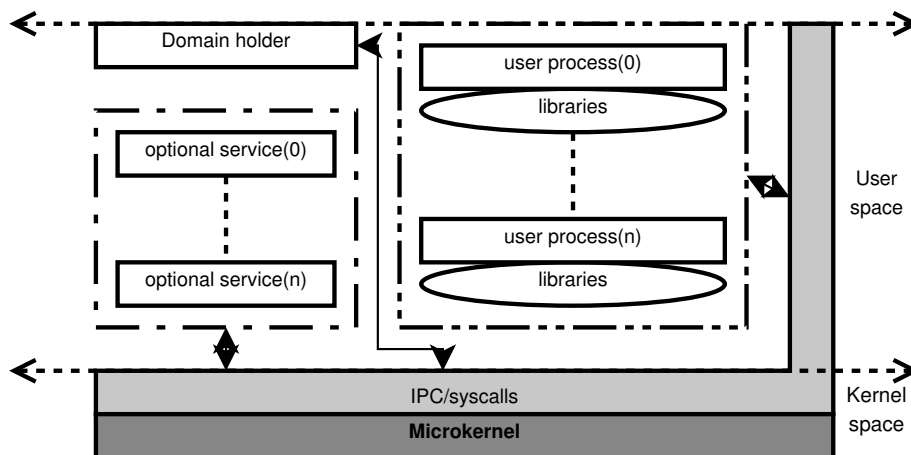


Figure 1.1: Abstract of system architecture design.

To implement abstraction layer on top of IPC, Jari OS provide a nice abstractions and instruments that makes development ease:

- IPC box abstraction (is a part of the system library that provide one universal access to the IPC functions).
- sbuf - universal abstraction to get access to the data (the data might be located in IPC buffers, DMA, or elsewhere ...).
- RPCv7 is the new universal way to bind object's operations.
- IDL with IDL compiler to generate RPC interface on top of the above listed features.

Generally, the pure featureless system consists on the described bottom runtime parts, and, optionally, IDL tools. With this set you can freely develop the system for your needs. However, Jari OS platform goes with many additional components, services, libraries and applications to let you fully concentrated on your purposes.

Another important difference with other operating systems and platforms is a special and unique Jari OS abstraction: **domain**. Jari OS domain might be presented as a set of the following:

- Processes and limits linked to the domain.
- Domain namespace.
- Domain and its resources events and notifications.

Each domain is isolated from each other domain, but there are possibility to access from one domain to other through translator. Domain support might be turned off if it's not required.

Acting as a platform Jari OS provides the whole set of the layers required to create a complete operating system. This includes driver model, file system layer, terminals, network layer and so on. For the testing and development purposes Jari OS supports POSIX via libraries and additional components (services) to support functionality and UNIX IPC described in SUSv3 standard. Anyway, you may exclude most of the POSIX functionality disabling this additional components to build your own required environment.

1.2.1 Microkernel

μ String is a natural microkernel that keeps minimality principle.

In case of architecture design microkernel doesn't provide any types of abstractions that used for applications (files, pipes, etc). Also, kernel doesn't contain user space resources within task information structure, and regular calls for task creation going via services.

Microkernel provide the basic functionality to whole operating system:

- Bootstrap and initialization of CPUs;
- Memory management;
- Scheduler;
- Interrupts and signals handling;
- IPC (copy for small messages, mapping for the big ones);
- Low level task operations;
- Core domains support;

At the moment microkernel using the multiboot specification to get up and running. I.e. loader boot up the microkernel itself and the other modules (usually it's the image of the initialization file system and minimal set of the services). Our platform uses the ELF standard binaries, and in this case microkernel has a minimal support to run a statically linked service binary. However, support for the binaries implemented on the top of the microkernel.

Microkernel provided IPC build from the two objects: IPC port and IPC channel. IPC port is a server side routing that gets a messages and provide it to the task owned it. IPC channel used to communicate with IPC port.

IPC port has a queue of the messages. Access to the queue has a blocking manner - i.e. the owner of the port will blocks on the read from the message queues until no messages has arrived. To sort the messages queue has a several politics, one of that is a prioritized queue that puts messages of the most prioritized tasks more closely to the head of the queue. This allows to perform RPC calls from the important tasks more often than from other ones.

IPC channel is abstraction to access to the message queue, but it has two modes: blocked and unblocked. Usually, the task send a message will blocks until server will not reply, but if allowed, there are a possibility to send a non-blocking messages (pulse) to the ports that permits this type of the access.

Since microkernel doesn't support filesystem or block devices it provide support the special memory objects, that handle many operations with the pages. For example, this made for the file systems to support file mapping. On each page fault on this area (of no pages loaded) microkernel sends a request to the memory object owner (file system) with this event to map and load this page, this works for other operations such as sync, map (access control), unmap (reference counting).

To support operational within industry area μ String support realtime features such as guaranteed time of interrupt and events handling, full preemption and so on. You can find more information about it in the special chapter regards to the microkernel.

1.2.2 Domain holder

Domain holder is a special service that manage the system environment and handles high-level resources.

Basically, we can determine the general purposes of the domain holder service:

- Keep and track the processes of the domain;
- Manage and storage of the processes resources;
- Manage access controls of the domain's processes;
- Resolve resources;
- Provide domain's namespace resources and system events;

- Process management;

Domain holder is the root of the domain and its namespace, if you will disable domains support, Jari OS, however, will contain one domain - root one, that served by the first process in the system - it's holder.

Each process within domain assigned will always has a interactions with it's holder. To use the system each process should get its resources and credentials from the holder if it's going to be initialized from the binary image, if process has to be cloned domain holder get acknowledge about it, and if permitted, clone all resources, create appropriate events to the resources owners.

Also, domain holder controls the bandwidth of the RPC requests going from the domain processes (if such limits are used in the configuration). According to this information holder service can make a decision to limit requests or to kill the process, it was made to prevent DDoS type of attacks. Since resolve requests might be flows between system services it very important to keep track on this.

To have a consistency in the processes tree, resources tree and keep limits running - while process burn or die holder gets an event about it, and makes the appropriate operations.

Jari OS platform provide all its objects via file system abstraction, and holder keeps the information about linked file systems. All resolving requests going through this service. I.e. it's not possible to open the resource session without allowance from the holder.

Other, important purpose of the holder is a support of the binary image loading. Actually, holder loads ELF interpreter and pass all the controls to it. It avoids domain holder timeslice waste.

Platform is designed to create a number of the domains, but only one of it's roots has a full access and allowance to create, configure and destroy domain - it's a root domain holder. Each domain may has its own access policy, but only first (default) domain holder has a rights to manage domains itself.

1.2.3 General libraries

General libraries contain all the generic stuff for the system. Actually, standard C library contains most of them, but also extend it with optional functionality that may or may not present in your system (depends on configuration).

We can divide this functionality by the following:

- Syscalls wrappers;
- Holder RPC interface wrappers;
- General abstractions used system wide;
- Initialization specific code;
- Standard C functions;

This set allows to use all features of the platform. But the generic set of libraries doesn't provide all interfaces and implementation of platform extended set layers i.e. network layer or file system layer.

Actually this set covers only the basic functionality of the platform, and, however, contains POSIX functions wrappers. Most of this functions located in libc that was taken from the *nix world and was extended and ported to the Jari OS. But you will find many extensions to the standard C and POSIX functions.

1.2.4 RPCv7 and vnode abstraction

1.2.5 IDL

In microkernel and multiservice system most of operations is going through RPC calls, instead of monolithic kernels where you can find a hundreds of syscalls. It's because of actual system is distributed via many services, i.e. filesystem is the one service, device driver is the another one and network stack operation occurs on the third one.

The first approaches shows how the complex interface design and implementation might be. There are a very huge set of the RPC functions that require interface implementation, and if it will be done manually it will takes a lot of time to implement and debug. To avoid bugs within interface implementation, and to separate functions and interface Jari OS platform uses it's own interface description language (IDL) that generates appropriate source code of the interfaces freeing developers from this painful job.

Generally IDL operates with interface and it's types and functions. Each interface has a number of functions and interface related types described via special Lisp-like syntax. This language translated to the internal trees data structures and passed to the back-end that generates the source code according to the interface description.

In this case interface changes, or some IPC-related changes will not affect the general functionality implementation, and bug found in generated code can be fixed at once.

1.3 Platform layers

Like it was described early Jari OS platform has a several additional ready-to-use layers to avoid painful implementation of the operating system basics and let developer concentrate on his/her engineering and/or researching tasks. This layers makes a platform fully functional system to envolve your own needs.

The general layers are: driver model, intended to ease develop concrete drivers; filesystem layer for the filesystem porting/implementation; terminals layer to get access to the system throught devices you need; networking layer to operate with a system throught network protocols.

Actually, each layer is not a separate service, usually it's a set of services and libraries implements required functionality. Since our platform is a micro-kernel and multi-service based it contain many specific difference with regular monolithic system, by the way, most of them are hidden from the developer to make development process ease. But it's better to know how it works to avoid performance penalty with your implementation.

During project life the number of layers might be grow up in number, and will be described as soon as it possible. But this book will describe only core team supported layers. For those layers implemented separatly check out external documentation.

1.3.1 Driver model

1.3.2 File system layer

1.3.3 Terminals layer

1.3.4 Network layer

Chapter 2

μ String third-generation microkernel

2.1 Introduction

2.2 Memory management

2.2.1 Basics of operation

2.2.2 Memory objects

2.3 Interrupts, events and IPC

2.3.1 Interrupts

2.3.2 IPC

2.4 Scheduling

2.5 Access control

2.6 Syscalls

Chapter 3

General abstractions

3.1 Introduction

3.2 IPC abstractions

3.3 Data access abstraction

3.4 Data structures

Chapter 4

IDL

4.1 Introduction

This paper aims to describe the implementation of the ORB for objects in RPC of the Jari OS, as well as to describe the IDL used within the project.

Since Jari OS platform is a distributed system, it requires a more flexible and extensible methods for remote procedure calls and interactions in general.

CORBA standard was taken as the basis, since it the most versalite and suitable for the target platform.

To conform project targets specification was extended, but CORBA/ORB specification might be extended in future with required additional features as QoS. In this case Jari OS implementation doesn't extend it architectically.

4.1.1 Document version

This document is not completed at all. It will be extended in case of additions to the specification and/or features fixes.

Keep this paper up-to-date.

4.2 Basic concepts

From the basic view there are three major parts of design are exists:

- IDL (specialized DSL conforming to CORBA)
- IDL compiler (basically it will support C)
- Environment libraries: one for server, other for client

IDL will describe the interface for the remote procedure calls, format and some additional logic for request processing (QoS for example).

IDL compiler will process an output for client and server parts that will assemble requests, parse it and so on. Primary goal is a C support (since all libraries

and servers written on C).

Libraries will support a low level support for CORBA/ORB environments, there libraries aims to be a layer between platform specific and interface implementation.

The basic flow might be shown on the figure 1. To support identification

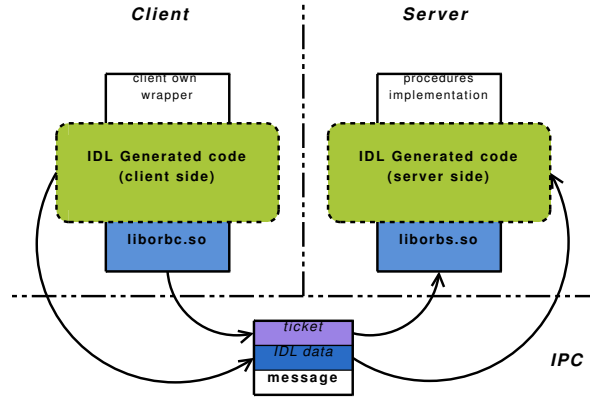


Figure 4.1: Abstract flow and basic diagram of the ORB design

and role recognizing of the caller and to be able to support additional features - new abstraction are added: “**ticket**”.

Ticket is the abstraction aims to describe client/caller identification and role recognizing. Actually is the abstraction structure for the low level implementation of role and security policy. By the way, it also used for internals of the ORB implementation.

Communications goes throw messages, and there messages is the abstraction too. Message itself is complete from the two parts:

- ticket
- message data

Message data may be used or maybe not by the IDL generated code, thus depends on the interface.

ORB environment is divided between library supporting code and IDL processed code. Library provide low level functionality and tickets workaround, interface protocol works provided via generated code.

IDL generated code for the specified interface might be called by the server code if it necessary.

The main objectives of such kind design are:

- protocol interfaces nesting
- flexible mechanism for the transfer of protocol messages
- messages QoS and water marks support

- messages forwarding and distributed processing support

This features are described below.

4.2.1 Protocol interfaces nesting

Protocol interfaces nesting is required by the various subsystems implementations such as filesystems, some networking subsystems (MODBUS, DeviceNet and so on). Client request goes to the one server usually, but it can be forwarded with some additions to other and so on. Typical flow can be shown on the figure 2. There are three stages shown on the diagram above:

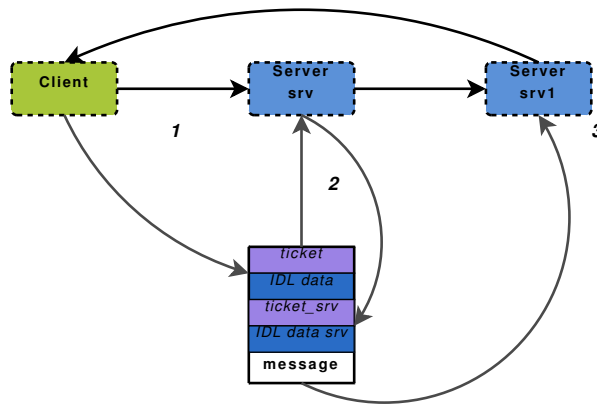


Figure 4.2: Abstract of interfaces nesting

1. Client sends the message to the server “srv” with its ticket and data
2. Server “srv” adds its own ticket and additional data to process on another server and forward the message
3. Server “srv1” process the message and reply to the client

The other common method of interface nesting is the mechanism for the message transparent transfer. It allows combine distributed and local message processing. CORBA environment implementation allows to call environment from the server code when it required. This allow server developer to implement an abstract design of the system overallly. For example, the scheme shown on the figure 2 might be changed – server “srv” and “srv1” may become libraries and works locally within one address space. In this case developer will not required to change the code base, since IDL might works in different cases.

4.2.2 QoS

The other side is support if the advanced features. Jari OS ORB implementation provide both functionality: **QoS** and **watermarks**. There are an optional features, but it highly required in case of some industry solutions.

For the domestic implementation IDL has support for separation via QoS labels, going through role identification, watermark, procedure call and other specified features.

CORBA-like environment (actually CORBA/ORB specification doesn't include QoS support) includes the ticket QoS label, that determined via specified rules. For example, to determine the QoS level in the figure 2 example Jari OS ORB implementation will take decision in case of two tickets or, if specified, only on one client original ticket. This made to provide support for more complex systems development and it's required by industry control and quality check systems as well as complex airport automatization systems.

QoS calculation has a predefined options:

- First ticket
- Compound ticket
- Caller ticket

Also, developer may describe it's own calculus of the QoS level within IDL syntax and definitions. It was made to support custom complex systems and it's semantic. First ticket policy works with the first taken ticket only. Compound ticket works in two modes, firstly ORB environment takes all tickets contains within message and takes more higher leveled, or, instead, more lowest tickets – depends on the mode chosen. Caller ticket policy always take decision on the caller original ticket.

By the way, depends on interface, several policy might be chosen for each interface in one ORB environment.

4.2.3 Distributed message processing

Distributed message processing is a feature that allows to separate request processing between several servers. That means - ORB implementation can split, forward, organize, substract messages.

There are several cases exist with distributed message processing, starting from the distributed filesystems, finishing with the complex requests in the area of the industry automation. To prevent desintegration of the interfaces Jari OS ORB implementation provide this feature to the domestic areas.

Messages may be processed in two modes:

- Linear mode
- Distributed mode

Linear mode is a simple case of distributed processing. This mode means that several servers works with one message in one by one character. This case shown on the figure 3. This example only for two servers shown, but this processing can be extended to *n* servers (determined in interface IDL description). To make a sense on it, there are three major stages:

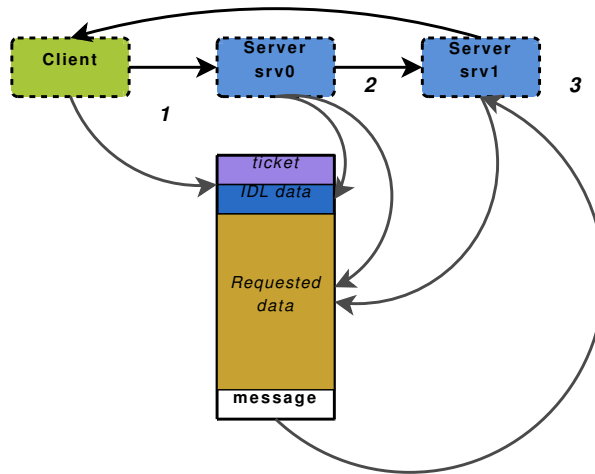


Figure 4.3: Linear mode of distributed message processing

1. Client sends the message to the server “srv0” with its ticket and data
2. Server “srv0” write a portion of required data, updates IDL headers and forwards this message to the srv1
3. Server “srv1” process the message, write portion of the data, reply to the client.

The decision of the reply made by the ORB environment and developer should not take care on it in this case. There are some note need to be made: ORB will calls user defined callback when message processing incomplete and should be forwarded to the next server, user-defined callback should return a server which will get a message next.

This scheme also possible for the centralized systems, where incomplete messages forwarding back to the central server that makes a decision of the next server. This is a typical load-balancing applications with a guaranteed i/o rates (if this will combined with the QoS feature).

Distributed mode is a more complex way to make message processing. This mode allows to divide message to the several ones (in the user-defined manner) and send it to the several servers. When all messages will be delivered ORB environment will complete the message to the original one and reply to the client.

This behavior is under research now, but it may be appear in the next ORB design implementation.

4.3 IDL

Usually, while you are developing a new server for the platform you have to solve several problems. You have to develop some protocol for interface between client and server. Also, you need to write code to pack the request on the client side and to unpack data on the server side, call to function, pack returning values back to the protocol interface and reply to the client, and again - parse data on the client side.

IDL provide help with protocol interface workaround. In the described case above all that you need is a function that will calls on the server side, other job will be made by IDL.

Jari OS uses its own IDL compiler that conforms to the CORBA standard – JICC (JariOS Interfaces C Compiler). It has a simple Scheme-like syntax and works above S-expressions. In Jari OS some specific types/definitions was added to the CORBA C language mapping. It's required to support additional features.

The basics elements of the IDL are:

- Interfaces
- Common types and constructed types
- Specific rules (extensions of Jari OS ORB implementation)

Interfaces describes the set of functions and its parameters for input and output.

Types describes what the data will flow between server and client.

Rules defines the optional and extended features of ORB implementation on IDL.

4.3.1 Interfaces

An interface description consists of at least one interface specification, that contains a set of functions desclarations.

Each declaration has two sets of data, that has attributes:

- in (passes to the function call)
- out (passes to the client)
- spec (specified by the interface and message flow semantics)

Like example, the simple one shown on the listing 1. In this example we defined a very simple interface, that will get type “int” from the client and reply with type “int” to the client.

4.3.2 Types used in interface descriptions

The JICC is support several types, it is a both: CORBA defined and internally used.

Listing 1 Very simple interface function specification

```

(define interface 'example
  (function hello((in (:int))
                  (out (:int))))
)

```

Integer types

The Jari OS IDL support the following integer types:

Type	Size	Value range
(signed) small	8 bit	-128 ... 128
(unsigned) usmall	8 bit	0 ... 255
(signed) short	16 bit	-32768 ... 32768
(unsigned) ushort	16 bit	-32768 ... 32768
(signed) int	32 bit	-2147483648 ... 2147483648
(unsigned) uint	32 bit	0 ... 4294967295
(signed) int64	64 bit	-9223372036854775808 ... 9223372036854775808
(unsigned) uint64	64 bit	0 ... 18446744073709551615

The CORBA IDL supports the following integer types:

Type	Size	Value range
short	16 bit	-32768 ... 32767
unsigned short	16 bit	0 ... 65565
long	32 bit	2147483648 ... 2147483648
unsigned long	32 bit	0 ... 4294967295
long long	64 bit	-9223372036854775808 ... 9223372036854775807
unsigned long long	64 bit	0 ... 18446744073709551615

NOTE: On the 64 bit platforms the CORBA IDL long type is always 32 bit.

Floating point types

The following types are supported:

Type	Size
float	32 bit
double	64 bit
long double (ldouble)	80 bit

Misc types

The following miscellaneous types are supported:

Type	Size	Value range
byte	8 bit	0 ... 255
void	undefined	N/A
(unsigned) uchar	8 bit	0 ... 255
char	8 bit	-128 ... 127
boolean	8 bit	true,false
char	8 bit	-128 ... 127
wchar	16 bit	-32768 ... -32767
octet	8 bit	0 .. 255

The **boolean** type is mapped to **char** type, 0 stands for **true** and other to **false**.

Platform specific types

Jari OS platform has a special set of types that may be used in the services.

Ticket type is a type to provide client identification to the service. There are several types exist:

- default, the first ticket will be given
- last, the last ticket will be provided
- array, all tickets will be provided (order: from first)

To specify the ticket type see below for examples:

Listing 2 Default ticket type

```
(define interface 'example
  (function hello((in (:int) (:ticket))
                 (out (:int))))
  ;; or use spec definion
  (function hello1((in (:int)
                     (spec (:ticket)))
                  (out (:int))))
)
```

The function that will be called will take “struct JILTicket” as argument in the cases above.

NOTE: Instead of the cases with one ticket, called functions will take the pointer to the “struct JILTicketArray”, that should be freed with IDL specified function (see below).

To freed the ticket array use the following function:

Listing 3 Last ticket type

```

(define interface 'example
  (function hello((in (:int) (:ticket 'last))
                  (out (:int))))
  ;; or use spec definion
  (function hello1((in (:int)
                       (spec (:ticket 'last)))
                   (out (:int))))
)

```

Listing 4 Array of tickets

```

(define interface 'example
  (function hello((in (:int)
                       (:ticket 'array))
                  (out (:int))))
  ;; or use spec definion
  (function hello1((in (:int)
                       (spec (:ticket 'array)))
                   (out (:int))))
)

```

Listing 5 struct JILTicketArray

```

struct JILTicketArray {
  int len;
  struct JILTicket *list;
};

/* to make your code inpedended from
 * the implementation use
 * macros below
 */

#define JILTicketArrayLen(n) (n)->len
#define JILTicketArrayList(n) (n)->list

```

Listing 6 JILTicketArray free function prototype

```

/*
 * free Ticket array.
 * It will set errno to EINVAL if pointer
 * or structure is
 * invalid.
 */
void JILTicketArrayFree(struct JILTicketArray*);

```

To learn more about ticket identification definitions read section “Ticket rules definition”.

NOTE: To learn about difference between “spec” define and pure “in” goto section “Specific types workaround”.

QoSLabel is a type used to provide information about QoS to your callback. There are no types of QoSLabel exist. QoSLabel may be specified with “spec” definition, or without it like it made with “ticket” type.

Listing 7 QoSLabel use

```

(define interface 'example
  (use QoS)
  (set-qos-policy 'caller-ticket)

  ;; functions
  (function hello((in (:int) (:QoSLabel))
                 (out (:int))))
  ;; or use spec definition
  (function hello1((in (:int)
                      (spec (:QoSLabel)))
                  (out (:int))))
)

```

To enable QoS feature you should define and describe it in the interface definition. See “Using QoS” section for more information.

Arrays

JICC is also support array types, here the example:

In the example above you will got the following C type for array:

```
int[10];
```

You may use any type for arrays, except specific ones, there are general form

Listing 8 Array definition

```
(define interface 'example
  (function hello((in (:int 'array(10)))
                  (out (:int))))
)
```

exist:

```
(:<type for array> 'array(<array size>))
```

Constructed types

You also may describe constructed types, i.e. structures.

To define constructed type you may use the standard form:

```
(define construct '<type_name>
  ((:<type> '<field_name>)
   ...
   (:<type> '<field_name>))
)
```

Constructed types should be defined within interface description, or within the general space.

Listing 9 Constructed type declaration

```
(define construct 'page_idx
  ((:uint64 'id)
   (:uint64 'addr)
   (:uint64 'offset)
  )
)
```

The following example will generate the following C structure definition:

Listing 10 Generated constructed type declaration

```
struct page_idx {
  uint64_t id;
  uint64_t addr;
  uint64_t offset;
};
```

Also, you may define C attribute for the structure, it can be made via the following standard form:

```
(define construct '<type_name>
  (set-struct-attr "packed")
  ((:<type> '<field_name>)
   ...
   (:<type> '<field_name>))
)
```

Type aliases

JICC supports type aliases for all types except platform specific. Alias define form is:

```
(set-type-alias '<alias_name>
  (<type or array>))
```

NOTE: You should also know that the following example:

Listing 11 Structure type definition

```
(set-type-alias 'page_idx_t
  ((define construct '_pg
    ((:uint64 'idx)
     (:uint64 'off))
    )))
```

Will generate the following C code declaration:

Listing 12 Structure type definition (Generated C code)

```
typedef struct _pg {
  uint64_t idx;
  uint64_t off;
} page_idx_t;
```

String type

String type is also supported by JICC. String type is a simple C-string with nil at the end. Space for the string allocating by the CORBA environment.

You may limit the length of the string. In this case you need to know that space will be allocated on the stack, and this size is limited to the 4096 bytes.

Passing data

To pass the data from client to server, that will be handled on the server side and not related to the RPC is a special type exist - "data". To use it use the following

Listing 13 String type declaration

```
(define interface 'example
  (function hello((in (:string))
                  (out (:int))))
)
```

Listing 14 String type declaration with limit declaration for the string length

```
(define interface 'example
  (function hello((in (:string 'array(128)))
                  (out (:int))))
)
```

Listing 15 Passing non-RPC data to the server

```
(define interface 'example
  (function hello((in (:data))
                  (out (:int))))
)
```

template: You should always keep in mind that this type should be always the last argument, otherwise you will get an error during IDL compilation.

To setup the data contents and data length there are special C type exist: This is used for both directions: to output data and to receive it on the client

Listing 16 Passing non-RPC data to the server, C

```
CORBAdata *yourdata;
void set_corba_data(CORBAdata *, void *);
void set_corba_data_length(CORBAdata *, ssize_t);
```

side.

4.3.3 Interface description

While you are designing interfaces you need to choose the interface backend and type of interface, this two features will impact the implementation.

This section describes interface description only, that will not affected with backend choose. The protocol itself isn't backend specific, while the backend implements the methods of message passing.

JICC supports the several methods to make intercommunications:

- Direct interface
- Indirect interfaces

- Distributed interfaces

Direct interfaces is a such type of interfaces that used in a simple cases while you need to make intercommunications between one server and clients. It doesn't support any type of redirections and it works in the classical client/server model.

In contrast, indirect interfaces provide support of the complex message passing between many servers and clients. It may be used in cases when you need to process the request on any server of your scheme.

Distributed interfaces provide mechanism for running client request on the several servers.

Usually, IDL creates a server loop (if this option is not disabled) that working in the following order:

- Receive a message
- Handle RPC information
- Call the required server procedure
- Reply to the message, or forward it (depends on the interface chosen)
- Return to the beginning

For the client side the wrappers will be generated that will pack the request data to the server. Generated code will depend on backend chosen.

To specify the interface type you should use "set-interface-type" function, by default the interface type is a "direct".

In addition to these methods of creating interfaces, there is another model to deal with IDL. The meaning of this model is to provide a more flexible mechanism of RPC messages handling. It's very often for OS system services to use its own code to deal with low-level IPC and IPC queues, in this case IDL may provide RPC headers handling only. To use this type use the "custom" interface part.

Direct interfaces

Direct interface working in the following manner:

- IDL takes the message from the backend IPC
- Call procedure related to the message
- IDL reply to the message with given values

To describe this type of interfaces use the following template: You may also use "spec" to specify specific types like "ticket". Generated code will call the procedure with a pointer to the message data and to the backend specific CORBA environment, but it will expect the error code and output data only. For more information see "Using generated code" section.

Listing 17 Direct interface template

```
(define interface '<interface name>
  (set-interface-type direct)
  (function <function name>((in (<parameters>))
    (out (<parameters>))))
  ...
)
```

Indirect interfaces

Indirect interfaces is more complex and requires called procedure to specify a point forward to. All manipulations goes in the backend independent way. Also, function may have additional parameters for passing. In the simple case this type of interface might be declared with the following template: It looks like

Listing 18 Indirect simple interface template

```
(define interface '<interface name>
  (set-interface-type indirect)
  (function <function name>((in (<parameters>))
    (out (<parameters>))))
  ...
)
```

a direct interface description, but generated code require from the procedure implementation to specify the point. If point pointer is "NULL" CORBA environment will reply to the callee, instead of forwarding.

If you need to forward the message with some additional data, you should describe it in the function declaration, like in the following template: Appended

Listing 19 Indirect simple interface template

```
(define interface '<interface name>
  (set-interface-type indirect)
  (function <function name>((in (<parameters>))
    (append (<parameters>))
    (out (<parameters>))))
  ...
)
```

data will be ignored if procedure implementation will not specify the point pointer.

Distributed interfaces

Custom interfaces

To use a custom interface, i.e. you will care about receiving, replying, forwarding messages you need to use the following template:

Listing 20 Custom interface template

```
(define interface '<interface name>
  (set-interface-type custom)
  (function <function name>((in (<parameters>))
    (out (<parameters>))))
  ...
)
```

It will unpack the input parameters and pack the output.

Set backend to use

Backend should be defined for all the interfaces you describe, it's a global description and used within the general context. To specify the backend to use use the function "set-backend", the syntax of this function is:

```
(set-backend "<name of the backend>")
```

The list of the available backends depends on your configuration and version.

By default "josipcbox" (Jari OS IPC box backend) is used.

4.3.4 Creating a complete interface

To describe the interface at all you will need to describe it in terms of IDL. This consists of setting up interfaces environment, including required components and so on. IDL have some global declarations and features that you need to know about.

To provide more easy interface development IDL comes with a simple pre-processor.

Interfaces description shall have the following order:

1. Preprocessor directives
2. Global settings for IDL environment
3. Types declaration (if needed)
4. Interfaces description

Developer also can use existing declaration and modules by the designed directives.

Connect modules and include statement

To use already defined components or modules there are two directives exist:

- “use”
- “include”

“use” directive used to include components from library, there components might be both types: provided internally by IDL or by it’s one of the components, or it may be external scheme-written module. Syntax:

```
(use <component name>)
```

Included components might have it’s own namespace, i.e. if you will declare component use within one interface it will apply to this interface only, but if you will declare it globally in your interfaces description it will apply to all of it. Internal components should be designed to avoid name conflicts, but if scheme-defined component name conflicts with internal IDL one, scheme component will be used.

“include” directive is used to include IDL scheme file to the interface description. Syntax:

```
(include "file name dot scm")
```

File search going first from the directory of scheme IDL description, after that it’s going in the path defined via arguments to the IDL compiler. This statement should be use globally only, it will ignored within the concrete interface description and error will be given. To point the search directory you also may use “set-search-path” directive, but to make effect to searching it should be declared before “include”(s). Syntax:

```
(set-search-path "path")
```

This path will be added to the list of search directories.

Macros

JICC preprocessor supports macros. Syntax:

```
(def-macro <macro-name> (<macro contents>))
```

Macros substitution works only for the first S-expression word, for example:

```
(def-macro foo (bar))
(foo foo(foo foo))
```

Will be processed as:

```
(bar foo(bar foo))
```

You can redefine macros, but you need to know, that macros doesn’t have namespace, you can redefine it elsewhere - it will be used everywhere.

NOTE: Special directives also may be redefined with macros.

4.4 Using the generated code

4.5 Backends

4.6 Compiler options

4.7 Examples

4.8 Build and install

4.9 Extend IDL

Chapter 5

Domains subsystem

5.1 Introduction

Since Jari OS going to be a platform for domestic industry areas we need to support different standard and conforms to support wide-range of industry standards running within one execution environment at the same time.

For this purpose platform intended to support new concept - **domain**.

Generally, domain is a set of processes that have specially determined limits, such as memory, CPU time and ability to acquire DMA, interrupts and other low level stuff. In addition to the listed limitation above intercommunication and access from one domain to the other domain is limited to, according to the domain configuration.

Basically Jari OS might work without domains support, that means the whole system operate within one root domain. This was made to exclude performance penalty on the operating systems that doesn't require this feature.

5.1.1 Used terms

In this documentation will be used to following list of terms:

- **Domain holder** - the main process of the domain that manage domain resources and tasks.
- **Namespace translator** - special process that allows to translate RPC requests between different domains namespaces.
- **Mirrored filesystem** - special filesystem that translate filesystem from one domain namespace to another and support local (and runtime) changes on it.
- **Security domain** - the set of the domain's processes without special attributes and privileges that allows direct intercommunication between domain's namespaces.

- **Translated resource** - resource shared between one or more namespaces through namespace translator.
- **Local system bus** - events and system-wide intercommunication bus for one domain.
- **Wide system bus** - events intercommunication bus operates above all existing and running domains.

5.1.2 Use cases

Many of industry standards require to separate execution environment within one running operating system, this creates many difficulties, and requires to create more complex implementation of the operating system internals.

Also, to decrease costs of the operating platform development for those systems that should contain different running execution environment Jari OS offers possibility to get it running with different namespaces. That means you may have run POSIX execution environment for your cross-platform applications in one domain and some other special execution environment for your platform specific applications/drivers/network stacks in another domain, and communicate between on one system.

Namespaces allow to prevent mistakes or security bugs on one namespace apply to the other namespace. For example, you may run your supply POSIX software that may contains security bugs in one domain, and your high priority servers and real-time software on another at the same system, and in this case you can be sure that security bugs in POSIX software will not corrupt operational of the other domain.

By the way, domains allow to create such environments, but it requires to configure it properly.

5.1.3 General overview

Domain itself is a set of processes running on the system allowed to communicate between each other directly. Where each domain has its own namespace.

Domain has an ID, default limits for memory and CPU time usage, but also can contain limits for processes amount.

Each domain has the representation in microkernel - this is a special structure that contains of the info about limits (only for resources that can be allocated through microkernel) and short domain name ID. In addition, each domain structure in the microkernel contains the list of allowed actions such as DMA allocation, interrupts handling and so on.

On the higher system level to support domain operation there are one domain holder exist - this is separate process that hosts the general request from domain processes, communicate with wide system bus, runs translators. This process contain all resources, limits information and other system related stuff assigned to the domain.

To intercommunicate between domains, translate other namespace contents there are translators exist. Translators have rights to communicate via IPC with other processes from other namespaces, but translator can be run by the domain holder only.

That means domain set has several special processes (ran by the trusted system part) with a special attributes, other processes will never have it.

Other processes is going to be a separate security group that controlled by the domain holder and translators.

The objects to translate is the following type of resources:

- System bus line
- Regular file or whole directory
- Special device file, or IPC related file (fifo, socket, shared memory, lock)
- Filesystem itself

Translators operates with destination objects directly, but translator doesn't provide a direct access to the object. That allows to limitate bandwidth of requests from domain.

5.1.4 Advantages and disadvantages

Domain concept was born after research with the related concepts, mostly with Plan 9 from Bell Labs namespaces. But Jari OS platform has the different architecture design compairing with other systems researched. The most common problems of the microkernel and multi-service platform are - the amount of context switches, complex architecture design, a high cost of synchronization and memory footprint. On the view it's going for performance degradation and possibility to make DDoS attacks to the services.

To conform for the new domain concept Jari OS architecture was changed: most common and general system services was splitted to one and assigned to serve one domain - actually become domain holder. To prevent DDoS attacks platform acquire new type of intercommunication - translating. This is the edge between flexibility, performance, complexity, stability and security.

Uncommonly, we will list disadvantages first:

- Translating performance penalty: while you translating request to the resource you need to forward request to the translator first, this increase the chain and amount of context switching;
- Complex domains configuration;
- Splitting general services functionality increase probability of errors within one address space;

On the other side we have the following advantages:

- Splitting general services reduce number of context switches and avoid complex synchronization between services. This feature will give a performance advantage;
- Separating processes with domains may prevent DDoS attacks (depends on configuration);
- Domains gives a very flexible security politics, translating only allowed resources may prevent a number of security holes;
- System can run a different execution environments at the same time;
- Domains gives an excellent advantages for building distributed environments;
- System architecture with support of this feature allows to conform to a number of standards, that make the platform is very flexible;
- Domains and translators doesn't require a very complex implementation, but provide the same features as existing ones (NFS, ACLs, etc);

Well, you may find other advantages and use cases for this great feature.

However, you may exclude domain feature and use only one root domain, that will take off all disadvantages listed above.

5.2 Implementation overview

Implementation of the domain subsystem mostly related to the microkernel services side, but some support required from the microkernel side.

Jari OS microkernel (μ String) development keeps the right way: less specific semantics and avoid services or drivers functionality in the kernel itself. To keep going on this way we should exclude higher level semantic from the kernel space. However, microkernel works with IPC, memory, CPU time, hardware resources (irqs, DMA, PIO ...) and to control the access microkernel must contain the minimal abstract rules about this. We cannot handle this type of checking to the userspace service due to the performance degradation, security reasons and growing up possibility of DDoS type attacks. On other hand we shouldn't load domain relations and translated resources to the microkernel, otherwise, we can provide it on the higher level.

Most common domain features such as layout tree, translation, inter-namespace messaging and events, implemented on the higher level. Root domain will always has a full set of access rights (capability to allocate all the microkernel resources, advanced rights to create, destroy and setup new domain, full IPC access). On other hand, each domain holder can run translator for it's own served domain. Domain limits, allowed capabilities and special attributes is going from the parent process, but can be changed with the following rules:

- Increase security rights and limits (the threshold determined by the general namespace limits and rights) may be done only by root domain holder;

- Decrease limits and remove capabilities might be done by the current domain holder;
- Nobody, except domain holder can assign translator flag to the process.

Each domain has its own default limits that assigned while domain creation process. There are one rule: domain holder (except root holder) cannot increase given limits.

We can differ limits: domain global, default per process assigned limits.

Global domain limits are limits for memory and CPU time usage given for the all processes working within one domain. Default per process limits are limits given by default to each domain process, each domain has its own default limits.

5.2.1 Initialization

If domains are used than root domain holder boot up other domain holders. Initialization doesn't require additional services to get domain up and running.

Root holder works with configuration data and works with the following order:

- Load domains configuration;
- Run each domain one by one, the order depends on the configuration, i.e. if domain A require translate some special resource from domain B, domain B will be booted first. But at the same moment we can load other domains simultaneously;
- Waits while all domains will be up and running, and mark the whole system up and running state;

Domain holder doesn't load anything except other domain holder, translators, services and daemons of the other domain will be started by the new up and run domain holder (translators) and other special specified service or other custom software.

The running of the new domain can be described by the following general procedures:

- Create new entry about domain in the microkernel side;
- Set limits, capability and other related attributes to the newly created process;
- Execute domain holder;

Each domain since it going to be initialized, trying to run other processes (according to the domain configuration). This procedure is going with the following:

- Run all translators first (if exist);

- Waits while all translators will be up and running;
- Execute other services or other software, according given configuration;

When all is up and running holder will send a notification to the root domain about it. On fail, the appropriate event will be signalled.

If domain cannot be run or contain errors, root holder will kill domain.

5.2.2 Microkernel-side support

Microkernel (μ String) contain a very basic support for domains. Kernel level side implementation of domain feature keeps the minimality as it possible.

Microkernel serve IPC, memory, CPU time and other low level resources, and, in this case, kernel should have namespace-related information about processes. Generally microkernel keeps the following domain related information:

- Domain ID;
- Defaults domain limits;
- Domain shortname;
- Reference count;

To deligate and separate processes from different domains - microkernel keeps the related information. I.e. each task has the following special attributes:

- Domain ID task belong to;
- Pointer to the domain structure;
- Special domain limits;
- Information about domain holder;
- Translator flag;

Microkernel is intended to make some mandratory checks for intercommunication between processes from different domains and a set of operation on the domain related attributes. There are a list of general rules:

- Process without translator flag cannot initiate connection to the process from the other domain;
- Translator flag might be set up only by domain holder;
- Process domain might be changed only by root domain holder;
- Domain limits might be setup only by root domain holder;

5.2.3 User-space support

5.3 Using name spaces

5.3.1 Default root name space

5.3.2 Multispace system

5.3.3 Configuration

5.4 Benchmarks

Chapter 6

Filesystem layer

Chapter 7

Driver model

7.1 Overview

This model is perceived to provide a general approach and needed abstractions for driver development. In the scope of the JariOS ideology drivers typically work as standalone device services. Hence, the model's major goals are:

- abstraction from IPC communications;
- uniform mechanism for processing logic separation from a protocol;
- support of multiple IPC mechanisms and command protocols for a single driver, and high flexibility relating this stuff;
- uniform way for device probing, registering and control.

The standard device service organization is depicted in the following figure. Server layer is a component which is intended to receive client messages, reply to them and manage client sessions. To provide communication with clients it creates and manages set of IPC boxes which can imply absolutely different mechanisms. Device manager is a component which knows in what manner devices should be used. It loads drivers, probes device interfaces and devices, and controls them. Moreover, it registers files for found devices and performs all needed initialization actions on subsystem level.

Typically such a manager deals with devices having familiar functions. In case of simple devices one can quite do without this component. Protocol layer performs function of client RPCs parsing and can provide for clients one or more protocols. Generic layer is intended to perform some high level processing defined by device logic functions and its usage scheme. This layer is not mandatory just as the device manager. Eventually, the component marked as 'drivers' on the figure is set of middle and low level drivers which implements a device command protocol and controls immediately hardware. As a matter of fact, low level drivers are not obligatory part of the same service. In case of hardware buses oriented

to devices with quite unrelated functions the drivers can be part of another standalone service.

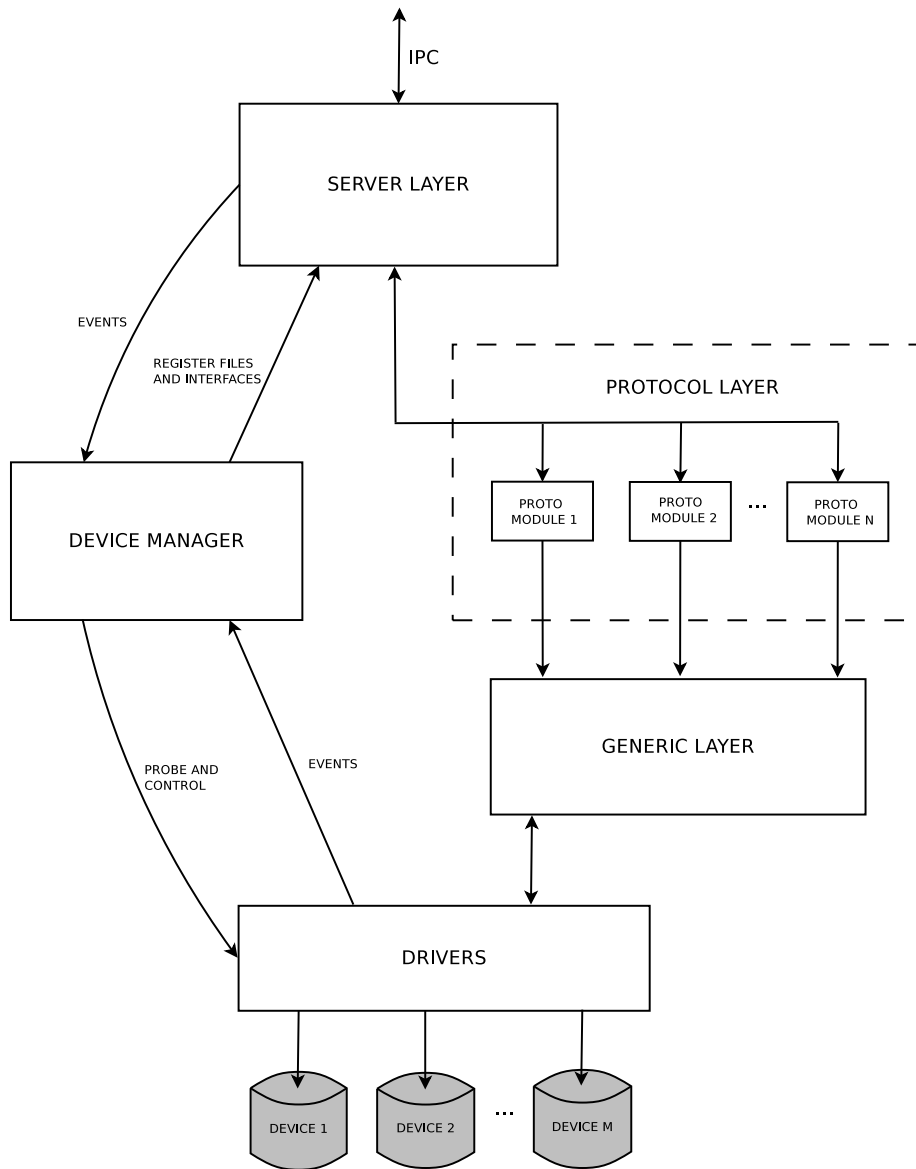


Figure 7.1: Device service organization

7.2 IPC classes

IPC class is essence determining an IPC mechanism which is used for communicating with clients. It includes a method creating IPC boxes and box service policy. Each device service can agregate many IPC classes as the discretion of its designer. Such an approach allows to achieve the goal relating IPC abstraction. A client can even run on a remote host, and other service parts are not obliged to know this.

7.3 Device class protocols

Device class is subset of devices united in single group due to similar logical functions, data flow control way, the same supported protocol, etc. Such a class implies some common functionality delivered to clients. Single physical device can quite belong to different classes. On RPC level each class delivers one or several protocols. Typically one class protocol concerns to one device class. In non so frequent cases some types of clients can require a different data flow control way so that to increase performance, for example. On the other hand, such a way can be allowable only for system services, not for usual applications. In such cases the class can deliver a separated protocol with its own set of RPCs. Each protocol can be accessible through one or all IPC mechanisms. The driver designer decides this. All devices without exception support a default protocol used at least for opening, closing and some notifications from another system services. After opening a client can enumerate class protocols and then switch to that one which is interesting for it. At a time only one protocol module can be active for each client.

The fig. 2 illustrates these manipulations.

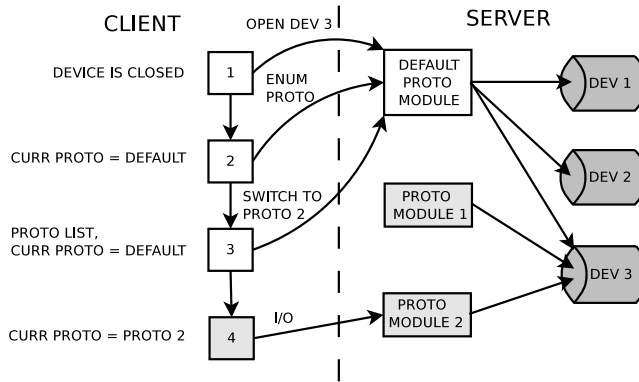


Figure 7.2: Using different class protocols

7.4 Device interfaces

In runtime all device instances exist in context of some device interface. This abstraction can be considered from different points of view depending on device type. For physical devices it represents a hardware interface the devices are plugged to. For physical devices which low level drivers execute in context of another service it corresponds an endpoint that data flow will directed to. Eventually, for virtual devices it represents a software abstract interface uniting subset of such devices by some criteria.

7.5 Device manager

As it was already noted, device manager is such a component which defines usage scheme of devices. A device can be used separately or can be part of some system consisting of several devices used in common. A driver should not care about that. Device manager assumes all need initialization, and a special more high level driver assumes respective processing in case of composite device usage. As well, it's implied to shift all server and file related initialization routine to the manager. Thus, in most cases middle and low level device drivers will not do anything unrelated to device control.

One should emphasize there is not some universal component of the system managing all devices without exception. Each device type should have its own manager if this component is needed for it at all. Different device types require reach set of functionality implementing different RPC protocols, providing different intermediate layers for request control, and so on. Hence, approach with universal manager could lead to too complex design hierarchy and system tuning.

To provide more flexibility device drivers can be dynamically loadable. One the one hand, it delivers from loading all drivers at the same time even if they

are not used. On the other hand, it delivers from source code modification at a new driver adding. The manager take that loading upon oneself.

Typical sequence of actions for a device manager is listed below:

1. device server creation;
2. IPC classes and class protocols registration;
3. loading of predefined set of drivers;
4. initiation of interfaces scanning and device probing;
5. file registration for all found devices;
6. events waiting; needed drivers loading at device hotplug, resource cleaning at device removing.

7.6 Middle and low level device drivers

The major function of middle level drivers is to implement command protocol used to communicate with devices. It means the driver should translate an interface independent request done by higher layer to a command which will be then issued to the device. The next function of the drivers is to implement the interface delivered to the device manager. It should provide the following set of functionality:

- scanning of hardware interfaces presented in the system
- interface probing
- interface state management
- device probing
- device state management

Low level device drivers are intended to provide device control at the physical layer. They deliver a command composed by the middle level driver by operating immediately with registers of the device or with registers of the respective bus adapter to a port of which the device is plugged.

7.7 Event model

When some device related event occurs in the system, the device manager receives a notification about it. It supplies two means to receive them: through system bus and through notification queue. Notifications through the system bus are used for devices whose low level driver executes in context of another service. These type of notifications is received from the system bus daemon. Notifications through the queue is applied when the low level driver belongs to the same service. As an event is typically detected in interrupt context, deliverence through the system bus is not appropriate in the case because the driver should send a message to the system bus daemon and block, thus delaying pending interrupts processing. The tab. 1 enumerates this type of events.

Table 7.1: Device events enumeration

Name	Description
NOTIFY_DEVICE_ADDED	Sent when a new device is plugged
NOTIFY_DEVICE_REMOVED	Device is detached from the interface
NOTIFY_DEVICE_STATUS_CHANGED	Applicable at internal device state change leading to another status reported by the device.

7.8 Configuration parameters

Each driver, device interface and device has set of named parameters reflected to devio file system and used for configuring. When a client wants to tune driver, interface or device work mode, it should simply open the respective file on devio file system and write to it an appropriate value. One file corresponds to one parameter.

Chapter 8

Terminals layer

Chapter 9

Network layer

Part II

Platform implementation

Chapter 10

Domain holder

10.1 Introduction

Domain holder is a second core component of the platform (the first is a microkernel) and it should be strongly well designed. As it was described in the first part of this book, carrier is a highly loaded service that process system requests. That mean the service should conform the following rules:

- Should has a simple and clear implementation;
- Should has a small amount of code lines as it possible;
- In case of a big set of running threads carrier implementation must be well adapted for concurrency execution and data accessing;
- Holder must have mecanismes to prevent attacks and overload cause by the system events;

Internal data is highly connected to each other and it goes to be very important to design and implement data accessors with a minimal set of locks and prevent threads to fall asleep on the locks. To be short: holder has the several functions loaded:

- Tracking of the tasks and task's resources;
- Handle system events;
- Security/limits checking and tracking;
- Locate resources within file system points;
- Provide system accounting information;
- Manage microkernel services (i.e. restore critical services if required);

Since this service process many resolve and system events it should be fast, however it's not safe. There are several possibilities to overload this service and make the system unreachable. To avoid this problem several ways exists.

10.1.1 Preventing overload

To prevent DDoS attacks and overload on any system event domain holder has several techniques. Firstly we will divide the possible sources of the invalid (or extremely high) load:

- System events such as filesystem unexpected shutdown;
- Invalid client code (special code that cause DDoS);
- Floating resolve requests;

Generally, we can run each group of tasks in different domains, but it will not solve the problem within one domain. Especially for this type of problem, Jari OS microkernel has prioritized message queues, support for the low level limits. But actually it doesn't solve it fully. The next problem is a floating resolve requests, due to the IPC message passing scheme there are a possible situation when many of thread will fell asleep on the resolve requests while waiting reply from the concrete file system service, but microkernel has a message forward feature that solve this and offer a better flexible mechanism for the resource resolving. However this microkernel-level solutions cannot solve the problem globally. For example, there are a possible situation while you may have an extremely amount of forwarding messaging within domain services, or your "normal" tasks will waits for a long time at the moment while the potential DDoS attackers loads the system with the same priority, another colorful example is restoring filesystem - when you will need to restore all sessions that was located on the dead copy and so on ...

To solve described problems domain holder support two techniques that intended to be a killers of the described situations above:

- DDoS heuristic analization;
- Request rate limits;
- On-demand sessions restoring;

The first two methods solve the problem with DDoS attacks, since domain holders knows about requests and it's kind calculate and analyze it's amount and validity. Creating limits for the unprivileged users will avoids extremely large number of floating requests.

With on-demand restoring mode we will avoid system overload while restoring sessions.

10.1.2 Used data structures

To increase performance carrier should operate with appropriate data structures offers appropriate search time and applicable memory footprint.

This service stores the very different type of information and store cache for resolve request. It's going to be possible (to keep the cache), since we're serving events with resources.

In this case due to the investigation and research service using the different data structure to store and access data.

To keep the track of the task structure and task's resources we provide two different binary search tree:

- Red and black search tree to keep tasks entries (with one complex non-binary tree that intended to solve parent-child relations);
- Splay tree for task's resources (it's a self balancing tree where most frequently accessed nodes are closely to the root node);

To store opened and evented resources we're using B+tree like a high performance lookup tree. This decision was made due to the research and comparison between B-tree and Bx-tree. The last one was rejected due to the complexity and unrelated target.

Resource points (file system mount tree) are stored in the non-binary graph, where lookup going through parsed directories.

Actually holder doesn't provide the real-time lookup due to the unlimited level and symbolic link support, also it cannot be provide due to the general file systems nature.

For current resource resolve transactions we're using splay tree like more appropriate structure to approach our needs.

10.2 Task and resources operational

10.2.1 General overview

One of the core job processing on the domain holder is a keeping track of the existing tasks within it's served domain. Microkernel provide low-level control machnismes to get the guarantee that holder will always know about all the tasks working in terms of it's own domain.

Holder always subscribed to the following tasks events:

- Creation (actually task cannot clone itself without holder permission);
- Termination;

In addition all events such as execution (changing image of the process) going throught domain holder. For sure, domain holder doesn't control the threads amount, however holder is able to get/set such limits.

In Jari OS some of the process resources actually controlled by the microkernel:

- Microkernel IPC objects;
- Memory;
- CPU time;
- Threads;
- Hardware resources such as DMA and other hardware architecture specifics (if exists);
- Interrupts;

But anyway, limits for this resources, changes and other type of its management going on the user-space throught domain holder.

Other resources such as:

- UNIX IPC objects;
- Files, file mappings and file descriptors;
- Network sockets;
- System events;
- Devices;
- Terminals;

Implemented outside of the microkernel and might be fully controlled by the holder or supplementary services (if configured).

That means that it's really possible to implement a well strong technique to control task permissions and resources, but it will require the well designed data storage with well defined concurrency access methods.

10.2.2 Tasks and resources storage

As described early we're using binary search tree to keep track about resources and task with hierarchy. Actually, each task is a complete structure that has set of data required to manage and support task in terms of the high level resources.

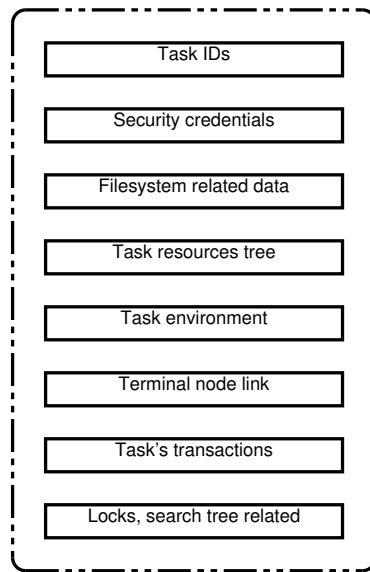


Figure 10.1: Task entry

On the figure bottom you can find the simple diagram shows the general task entry. There are the set of important task information needed to operate with both low level and high level abstractions.

Task IDs is a set of identifiers of the task, and it's parent information. Also this identification contain domain special attributes if platform used with domains.

Security credential is a set of the data that points to the security information i.e. data related to the supported security model and limits information.

Filesystem related data is a set of information regarding to the filesystem resolve process, current directory, binary image name and so on.

Task resources tree is a special binary search tree that contain links to the opened resources such as sessions and events.

Task environment is a special customized structure that offer a very flexible mechanisms to keep specific environment variables, additional data that will conforms to the supported environment API. For example, if your environment is POSIX-compatible this structure will keeps environment variables.

Terminal node link intended to appear in systems with terminal support. For example some type of POSIX signals connected to the task's running terminal events. However, if your system doesn't need to support terminal this data field will be absent.

Task's transactions is another binary search tree (splay tree) with links to the task's current running transactions.

Also this structure has locks, and special data storage entries to provide more performance while operate with this structure.

10.2.3 Tasks hierarchy storage

As it was described early tasks hierarchy stored with a two methods: linear (just to keep track of the running processes in the system, but with IDs of the parent), and as a graph showing real relations as it shown on the figure below.

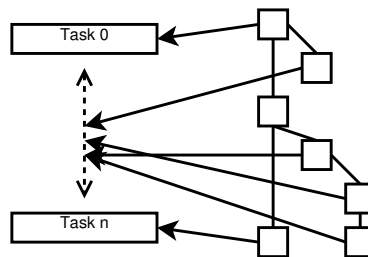


Figure 10.2: Tasks hierarchy storage

This scheme allows to operate with this information more flexible avoiding data duplication. Most of the tasks can be made through this two different search structures.

10.3 Filesystem points operational

10.4 System bus operational

10.5 Domains

10.6 Techniques used to prevent DDoS attacks

10.7 Translators

Chapter 11

Filesystem layer

11.1 Introduction

11.1.1 Network filesystem

11.1.2 Distributed filesystem

11.2 General vfs library

11.3 Remote procedures

11.4 Filesystem unions

11.5 Filesystem container library

11.6 Block resources data access library