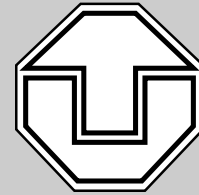


**TECHNISCHE UNIVERSITÄT
DRESDEN**



Fakultät Informatik

**Technische Berichte
Technical Reports**

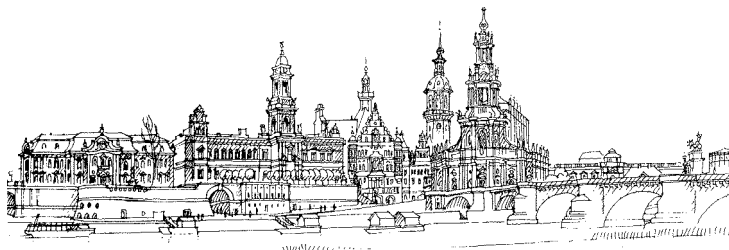
ISSN 1430-211X

TUD-FI06-07 Dezember 2006

**Norman Feske and Christian
Helmuth**

Institute for System Architecture, Operating
Systems Group

**Design of the Bastei OS
Architecture**



*Technische Universität Dresden
Fakultät Informatik
D-01187 Dresden
Germany*

URL: <http://www.inf.tu-dresden.de/>

Design of the Bastei OS Architecture

Norman Feske and Christian Helmuth

January 4, 2007

In the software world, high complexity of a problem solution comes along with a high risk for bugs and vulnerabilities. This correlation is particular perturbing for todays commodity operating systems with their tremendous complexity. The numerous approaches to increase the user's confidence in the correct functioning of software comprise exhaustive tests, code auditing, static code analysis, and formal verification. Such quality-assurance measures are either rather shallow or they scale badly with increasing complexity.

The operating-system design presented in this paper focuses on the root of the problem by providing means to minimize the underlying system complexity for each security-sensitive application individually. On the other hand, we want to enable multiple applications to execute on the system at the same time whereas each application may have different functional requirements from the operating system. Todays operating systems provide a functional superset of the requirements of all applications and thus, violate the principle of minimalism for each single application. We resolve the conflict between the principle of minimalism and the versatility of the operating system by decomposing the operating system into small components and by providing a way to execute those components isolated and independent from each other. Components can be device drivers, protocol stacks such as file systems and network stacks, native applications, and containers for executing legacy software. Each application depends only on the functionality of a bounded set of components that we call *application-specific trusted computing base (TCB)*. If the TCBs of two applications are executed completely *isolated* and *independent* from each other, we consider both TCBs as minimal.

In practice however, we want to share physical resources between multiple applications without sacrificing their independence. Therefore, the operating-system design has to enable the assignment of physical resources to each application and its TCB to maintain independence from other applications. Furthermore, rather than living in complete isolation, components require to communicate with each other to cooperate. The operating-system design must enable components to create other components and get them to know each other while maintaining isolation from uninvolved parts of the system.

First, we narrow our goals and pose our mayor challenges in Section 1. Section 2 introduces our fundamental concepts and protocols that apply to each component in the system. In Section 3, we present the one component that is mandatory part of each TCB, enables the bootstrapping of the system, and provides abstractions for the lowest-level resources. We exercise the composition of the presented mechanisms by the means of process creation in Section 4.

1 Goals and Challenges

Our design process was guided by the vision to execute the following types of components in a secure manner concurrently on one machine:

Device drivers

Device drivers translate the facilities of raw physical devices to device-class-specific interfaces to be used by other components. They contain no security policies and provide their services to only one client component per device.

Services that multiplex resources

To make one physical resource (e. g., a device) usable by multiple components at the same time, the physical resource must be translated to multiple virtual resources. For example, a frame buffer provided by a device driver can only be used by one client at the same time. A window system multiplexes this physical resource to make it available to multiple clients. Other examples are an audio mixer or a virtual network hub. In contrast to a device driver, a *resource multiplexer* deals with multiple clients and therefore, plays a crucial role for maintaining the independence and isolation of its clients from each other.

Protocol stacks

Protocol stacks translate low-level protocols to a higher and more applicable level. For example, a file system translates a block-device protocol to a file abstraction, a TCP/IP stack translates network packets to a socket abstraction, or a widget set maps high-level GUI elements to pixels. Compared to resource multiplexers, protocol stacks are typically an order of magnitude more complex. Protocol stacks may also act as resource multiplexers. In this case however, high complexity puts the independence and isolation of multiple clients at a high risk. Therefore, our design should enable the instantiation of protocol stacks per application. For example, instead of letting a security-sensitive application share one TCP/IP stack with multiple other (untrusted) applications, it could use a dedicated instance of a TCP/IP stack to increase its independence and isolation from the other applications.

Containers for executing legacy software

A *legacy container* provides an environment for the execution of existing legacy software. This can be achieved by the means of a virtual machine (e. g., a Java VM, a virtual PC), a compatible programming API (e. g., POSIX, Qt), a language environment (e. g., LISP), or a script interpreter. In the majority of cases, we regard legacy software as an untrusted black box. One particular example for legacy software are untrusted legacy device drivers. In this case, the container has to protect the physical hardware from potentially malicious device accesses by the untrusted driver. Legacy software may be extremely complex and resource demanding, for example the Firefox web browser executed on top of the X window system and the Linux kernel inside a virtualized PC. In this case, the legacy container may implement sophisticated techniques such as virtual memory.

Small custom security-sensitive applications

Alongside legacy software, small custom applications implement crucial security-sensitive functionality. In contrast to legacy software, which we mostly regard as untrusted anyway, a low TCB complexity for custom applications is of extreme importance. Given the special liability of such an application, it is very carefully designed, low complex, and requires as little as possible infrastructure. A typical example is a cryptographic component that protects credentials of the user. Such an application does not require swapping (virtual

memory), a POSIX API, or a complete C library. Instead, the main objectives of such an application are to avoid as much as possible code from being included in its TCB and to keep its requirements at a minimum.

Our design must be able to create and destroy subsystems that are composed of multiple such components. The *isolation* requirement as stated in the introduction raises the question of how to organize the locality of name spaces and how to distribute access from components to other components within the system. The *independence* requirement demands the assignment of physical resources to components such that different applications do not interfere. Instead of managing access control and physical resources from a central place, we desire a distributed way for applying policy for trading and revocating resources and for delegating rights.

2 Interfaces and Mechanisms

The system is structured as a tree. The nodes of the tree are processes. A node, for which sub-nodes exist, is called the *parent* of these sub-nodes (*children*). The parent creates children out of its own resources and defines their execution environment. Each process can announce services to its parent. The parent, in turn, can mediate such a service to his other children. When a child is created, its parent provides the initial contact to the outer world via the following interface:

```
void exit(int exit_value);

Capability session(String service_name,
                  String args);

void close(Capability session_cap);

int announce(String service_name,
             Capability service_root_cap);

int transfer_quota(Capability to_session_cap,
                  String amount);
```

exit is called by a child to request its own termination.

session is called by a child to request a connection to the specified service as known by his parent whereas *service_name* is the name of the desired service *interface*. The way of resolving or even denying a *session* request depends on the policy of the parent. The *args* parameter contains construction arguments for the session to be created. In particular, *args* contains a specification of resources that the process is willing to donate to the server during the session lifetime.

close is called by a child to inform its parent that the specified session is no longer needed. The parent should close the session and hand back donated resources to the child.

announce is called by a child to register a locally implemented service at his parent. Hence, this child is a server.

transfer_quota enables a child to extend its resource donation to the server that provides the specified session.

We provide a detailed description and motivation for the different functions in Sections 2.1 and 2.2.

2.1 Servers

Each process may implement services and announce them via the *announce* function of the parent interface. When announcing a service, the server specifies a *root* capability for the implemented service. The interface of the root capability enables the parent to create, configure, and close sessions of the service:

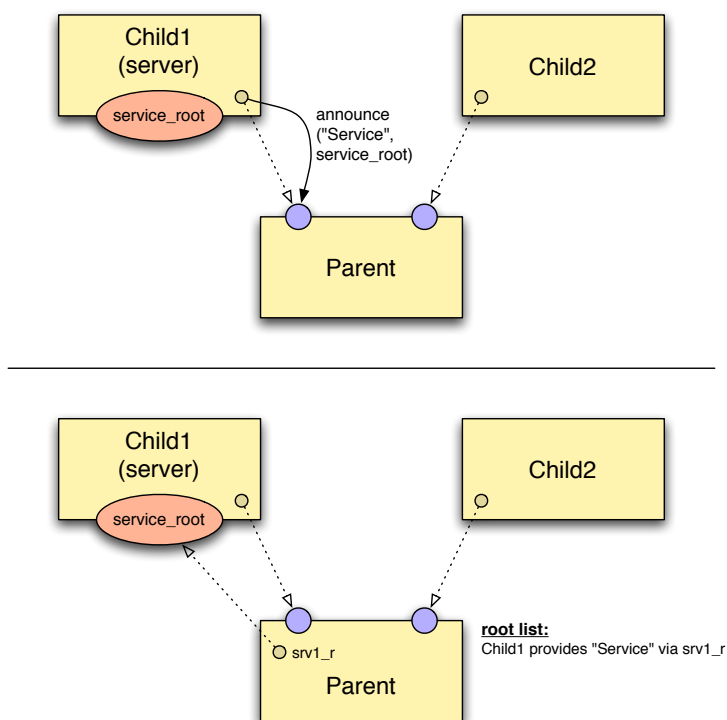


Figure 1: Announcement of a service by a child (server). Colored circles at the edge of a component represent remotely accessible objects. Small circles inside a component represent a reference (Capability) to a remote object. A cross-component reference to a remote object is illustrated by a dashed arrow. An opaque arrow symbolizes a RPC call/return.

```
Capability session(String args);

int transfer_quota(Capability to_session_cap,
                  String amount);

void close(Capability session_cap);
```

Figure 1 illustrates an announcement of a service. Initially, each child has a capability to its parent. After Child1 announces its service “Service”, its parent knows the root capability of this service under the local name `srv1_r` and stores the root capability with the announced service name in its *root list*. The root capability is intended to be used and kept by the parent only.

When a parent calls the `session` function of the root interface of a server child, the server creates a new client session and returns the corresponding `client_session` capability. This session capability provides the actual service-specific interface. The parent can use it directly or it may pass it to other processes, in particular to another child that requested the session. In Figure 2, Child2 initiates the creation of a “Service” session by a `session` call at its parent capability (1). The parent uses its *root list* to look up the root capability that matches the service name “Service” (2) and calls the `session` function at the server (3). Child1 the server creates a new session (`session1`) and returns the session capability as result of the `session` call (4). The parent now knows the new session under the local name `srv1_s1` (5) and passes the session capability as return value of Child2’s initial `session` call (6). The parent maintains a *session list*,

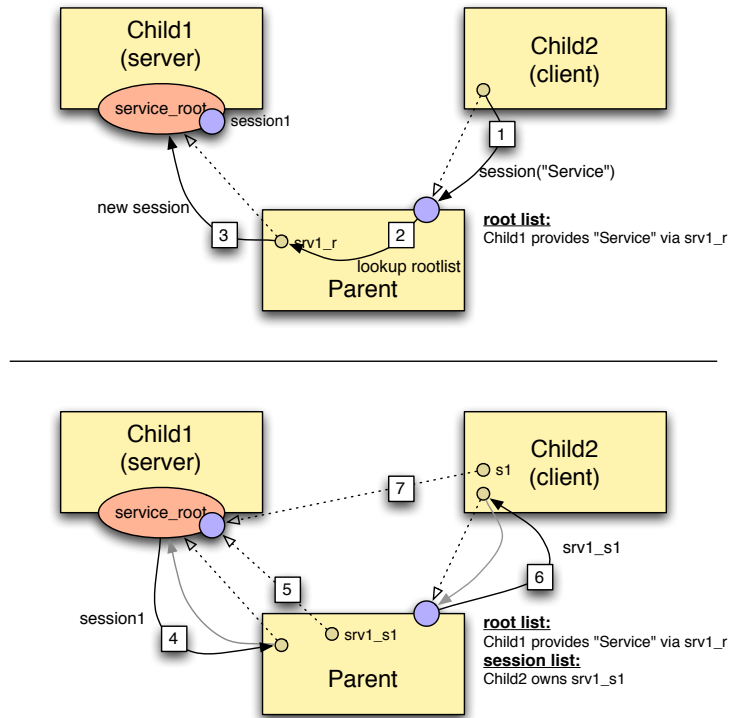


Figure 2: Service request by a client.

which stores the interrelation between children and their created sessions. Now, Child2 has a direct communication channel to `session1` provided by the server (Child1) (7).

The `close` function of the root interface instructs the server to destroy the specified session and to release all session-specific resources.

Even though the prior examples involved only one parent, the announce-request mechanism can be used recursively for tree structures of any depth and thus allow for partitioning the system into subsystems that can cooperate with each other whereas parents are always in complete control over the communication and resource usage of their children (and their subsystems).

Figure 3 depicts a nested subsystem on the left. Child1 announces his service named "Service" at his parent that, in turn, announces a service named "Service" at the Grandparent. The service names do not need to be identical. Their meaning spans to their immediate parent only and there may be a name remapping on each hierarchy level. Each parent can decide upon itself whether to further announce services of their children to the outer world or not. The parent can announce Child1's service to the grandparent by creating a new root capability to a local service that forwards session-creation and closing requests to Child1. Both Parent and Grandparent keep their local root lists. In a second step, Parent2 initiates the creation of a session to the service by issuing a `session` request at the Grandparent (1). Grandparent uses its root list to look up the service-providing child (from Grandparent's local view) Parent1 (2). Parent1 in turn, implements the service not by itself but delegates the `session` request to Child1 by calling the `session` function of the actual "Service" root interface (3). The session capability, created by Child1 (4), can now be passed to Child1 as return value of nested `session` calls (5, 6). Each involved node keeps the local knowledge about the created session such that later, the session can be closed in the same nested fashion.

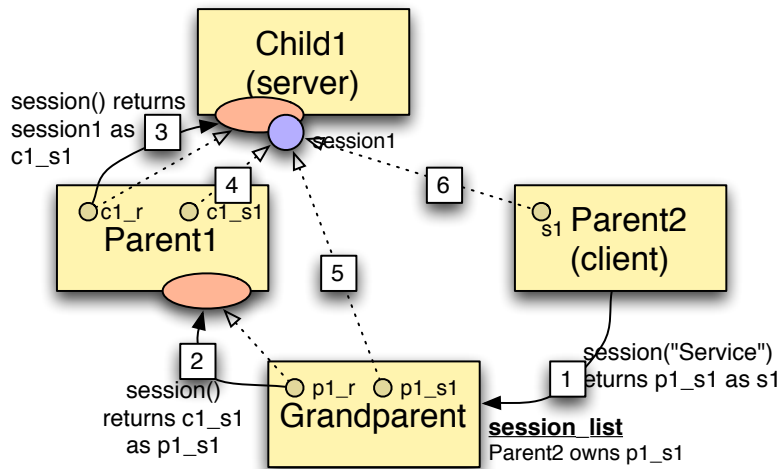
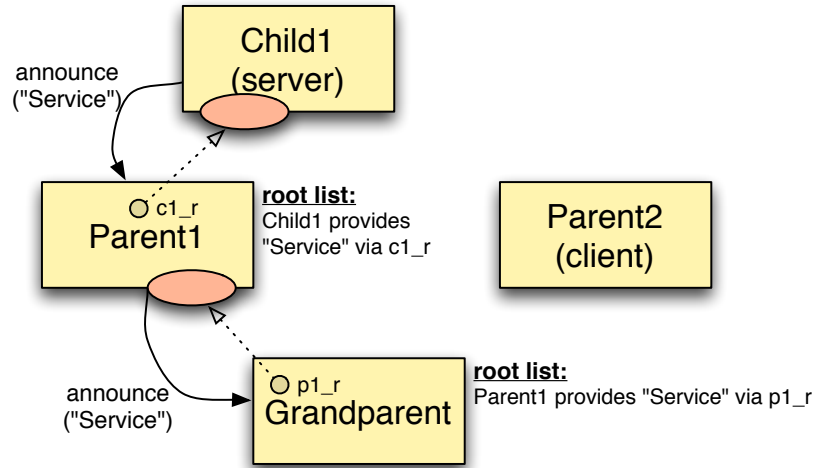


Figure 3: Announcement and request of a service in a subsystem. For simplicity, parent capabilities are not displayed.

2.2 Quota

Each process that provides services to other processes consumes resources on behalf of its clients. Such a server requires memory to maintain session-specific state, processing time to perform the actual service function, and eventually further system resources (e. g., bus bandwidth) dependent on client requests. To avoid denial-of-service problems, a server must not allocate such resources from its own budget but let the client pay. Therefore, a mechanism for donating resource quotas from the client to the server is required. Both client and server may be arbitrary nodes in the process tree. In the following, we examine the trading of resource quotas within the recursive system structure using memory as an example.

When creating a child, the parent assigns a part of its own memory quota to the new child. During the lifetime of the child, the parent can further transfer quota back and forth between the child's and its own account. Because the parent creates its children out of its own resources, it has a natural interest to correctly manage child quotas. When a child requests a session to a service, it can bind a part of its quota to the new session by specifying a resource donation as an argument. When receiving a session request, the parent has to distinct three different cases, dependent on where the corresponding server resides:

Parent provides service

If the parent provides the requested services by itself, it transfers the donated amount of memory quota from the requesting child's account to its own account to compensate the session-specific memory allocation on behalf of its own child.

Server is another child

If there exists a matching entry in the parent's root list, the requested service is provided by another child (or a node within the child subsystem). In this case, the parent transfers the donated memory quota from the requesting child to the service-providing child.

Delegation to grandparent

The parent may decide to delegate the session request to his own parent because the requested service is provided by a lower node of the process tree. Thus, the parent will request a session on behalf of his child. The grandparent neither knows nor cares about the actual origin of the request and will simply decrease the memory quota of the parent. For this reason, the parent transfers the donated memory quota from the requesting child to his own account before calling the grandparent.

This algorithm works recursively. Once, the server receives the session request, it checks if the donated memory quota suffices for storing the session-specific data and, on success, creates the session. If the initial quota donation turns out to be too scarce during the lifetime of a session, the client may make further donations via the `transfer_quota` function of the parent interface that works analogously.

If a child requests to close a session, the parent must distinguish the three cases as above. Once, the server requests to close the session, the child is responsible to release all resources that were used for this session. After the server releases the session-specific resources, the server's quota can be decreased to the prior state. However, an ill-behaving server may fail to release those resources by malice or caused by a bug.

If the misbehaving service was provided by the parent himself, it has the full authority to not hand back session-quota to his child. If the misbehaving service was provided by the grandparent, the parent (and its whole subsystem) has to subordinate. If, however, the service was provided by another child and the child refuses to release resources, decreasing its quota after closing the session will fail. It is up to the policy of the parent to handle such a failure either by punishing (e. g.,

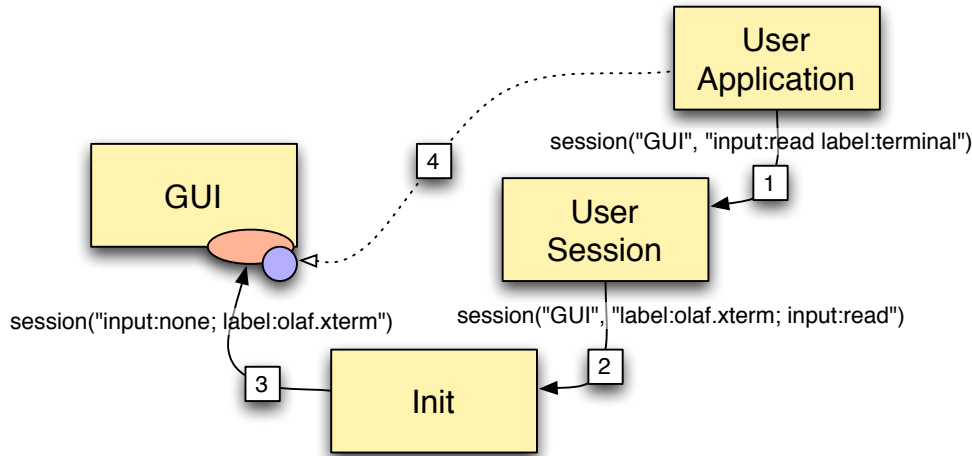


Figure 4: Successive application of policies at the creation time of a new session.

killing the the misbehaving server) or by granting more of its own quota. Generally, misbehavior is against the server’s own interests and each server would obey the parent’s `close` request to avoid intervention.

2.3 Successive policy management

For supporting a high variety of security policies for access control, we require a way to bind properties and restrictions to sessions. For example, a session to a file service would require to know the user identity and a group identity to enforce the Unix access-control scheme. On session creation, the `session` call takes an `args` argument that can be used for that purpose. It is a list of tag-value pairs describing the session properties. By convention, the list is ordered by attribute priority starting with the most important property. The server uses these `args` as construction arguments for the new session and enforces the security policy as expressed by `args` accordingly. Whereas the client defines its desired session-construction arguments, each node that is incorporated in the session creation can alter these arguments in any way and may add further properties. This effectively enables each parent to impose any desired restrictions to sessions created by its children. This concept works recursively and enables each node in the process hierarchy to control exactly the properties that it knows and cares about. As a side note, the specification of resource donations as described in the Section 2.2 is performed with the same mechanism. A resource donation is a property of a session.

Figure 4 shows an example scenario. A user application issues the creation of a new session to the GUI server and specifies its wish for reading user input and using the string “Terminal” as window label (1). The parent of the user application is the user manager that introduces user identities into the system and wants to ensure that each displayed window gets tagged with the user and the executed program. Therefore, it overrides the `label` attribute with more accurate information (2). Note that the modified argument is now the head of the argument list. The parent of the user manager, in turn, implements further policies. In the example, `Init`’s policy prohibits the user-manager subtree from reading input (for example to disable access to the system beyond official working hours) by redefining the `input` attribute and leaving all other attributes unchanged (3). The actual GUI server observes the final result of the successively changed session-construction arguments (4) and it is responsible for enforcing the specified policy for the lifetime of the session. Once a session got established, its properties are fixed and cannot be changed.

3 Core - the root of the process tree

Core is the first user-level program that takes control when starting up the system. It has access to the raw physical resources and converts them to abstractions that enable multiple programs to use these resources. In particular, core converts the physical address space to higher-level containers called *dataspaces*. A dataspace represents a contiguous physical address space region with an arbitrary size (at page-size granularity). Multiple processes can make the same dataspace accessible in their local address spaces. The system on top of core never deals with physical memory pages but uses this uniform abstraction to work with memory, memory-mapped I/O regions, and ROM areas.

Using only contiguous dataspaces may lead to fragmentation of the physical address space. This property is, however, only required by a few rare cases (e. g., DMA transfers). Therefore, later versions of the design will support non-contiguous dataspaces.

Furthermore, core provides all prerequisites to bootstrap the process tree. These prerequisites comprise services for creating processes and threads, for allocating memory, for accessing boot-time-present files, and for managing address space layouts. Core is almost free from policy. There are no configuration options. The only policy of core is the startup of the init process to which core grants all available resources to init.

In the following, we explain the session interfaces of core's services in detail.

3.1 RAM - allocator for physical memory

A RAM session is a quota-bounded allocator of blocks from physical memory. There are no RAM-specific session-construction arguments. Immediately after the creation of a RAM session, its quota is zero. To make the RAM session functional, it must be loaded with quota from another already existing RAM session, which we call *reference account*. The reference account of a RAM session can be defined initially via:

```
int ref_account(Capability ram_session_cap);
```

Once the reference account is defined, quota can be transferred back and forth between the reference account and the new RAM session with:

```
int transfer_quota(Capability ram_session_cap, size_t amount);
```

Provided, the RAM session has enough quota, a dataspace of a given size can be allocated with:

```
Capability alloc(size_t size);
```

The result value of `alloc` is a capability to the dataspace object implemented in core. This capability can be communicated to other processes and can be used to make the dataspace's physical-memory region accessible from these processes. An allocated dataspace can be released with:

```
void free(Capability ds_cap);
```

The `alloc` and `free` calls track the used-quota information of the RAM session accordingly. Current statistical information about the quota limit and the used quota can be retrieved by:

```
size_t quota();
size_t used();
```

Closing a RAM session implicitly destroys all allocated dataspaces.

3.2 ROM - boot-time-file access

A ROM session represents a boot-time-present read-only file. This may be a module provided by the boot loader or a part of a static ROM image. On session construction, a file identifier must be specified as session argument using the tag `filename`. The available filenames are not fixed but depend on the actual deployment. On some platforms, core may provide logical files for special memory objects such as the GRUB multiboot info structure or a kernel info page. The ROM session enables the actual read access to the file by exporting the file as dataspace:

```
Capability dataspace();
```

3.3 IO_MEM - memory mapped I/O access

With `IO_MEM`, core provides a dataspace abstraction for non-memory parts of the physical address space such as memory-mapped I/O regions or BIOS areas. In contrast to a memory block that is used for storing information of which the physical location in memory is of no matter, a non-memory object has a special semantics attached to its location within the physical address space. Its location is either fixed (by standard) or can be determined at runtime, for example by scanning the PCI bus for PCI resources. If the physical location of such a non-memory object is known, an `IO_MEM` session can be created by specifying `io_mem_base` and `io_mem_size` as session construction arguments. The `IO_MEM` session then provides the specified physical memory area as dataspace:

```
Capability dataspace();
```

There are further services required for full device-driver support outside of core. For example, services to program I/O ports and install IRQ handlers. These services are not specified yet.

3.4 RM - managing address space layouts

RM is a *region manager* service that allows for constructing address space layouts (*region map*) from dataspaces and that provides support for assigning region maps to processes by paging the process' threads. Each RM session corresponds to one region map. After creating a new RM session, dataspaces can be attached to the region map via:

```
void *attach(Capability ds_cap,
            size_t size=0, off_t offset=0,
            addr_t local_addr = 0);
```

The `attach` function inserts the specified dataspace into the region map and returns the actually used start position within the region map. By using the default arguments, the region manager chooses an appropriate position that is large enough to hold the whole dataspace. Alternatively, the caller of `attach` can attach any sub-range of the dataspace at a specified target position to the region map by specifying the optional arguments. Note that the interface allows for the same dataspace to be attached not only to multiple region maps but also multiple times to the same region map. As the counterpart to `attach`, `detach` removes dataspaces from the region map:

```
void detach(void *local_addr);
```

The region manager determines the dataspace at the specified `local_addr` (not necessarily the start address) and removes the whole dataspace from the region map. To enable the use of a RM session by a process, we must associate it with each thread running in the process. The function

```
Capability add_client(Capability thread);
```

returns a *pager* that handles the page faults of the specified `thread` according to the region map. With subsequent page faults caused by the thread, the address-space layout described by the region map becomes valid for the process that is executing the thread.

3.5 CPU - allocator for processing time

A CPU session is an allocator for processing time that allows for the creation, the control, and the destruction of threads of execution. There are no session arguments used. The functionality of starting and killing threads is provided by two functions:

```
Capability create_thread(const char* name);  
void kill_thread(Capability thread_cap);
```

The `create_thread` function takes a symbolic thread name (that is only used for debugging purposes) and returns a capability to the new thread. This capability refers to a thread object with the following operations:

```
int set_pager(Capability pager_cap);  
int start(addr_t ip, addr_t sp);
```

The `set_pager` function registers the thread's pager whereas `pager_cap` (obtained by calling `add_client` at a RM session) refers to the RM session to be used as address-space layout. For starting the actual execution of the thread, its initial instruction pointer (`ip`) and stack pointer (`sp`) must be specified for the `start` operation.

Future versions of the CPU service will provide means to further control the thread during execution (e. g., pause, execution of only one instruction), acquiring thread state (current registers), and configuring scheduling parameters.

3.6 TASK - providing address spaces

A TASK session corresponds to a memory protection domain (*task*). Together with one or more threads and an address-space layout (RM session), it forms a process. There are no session arguments. After session creation, the task contains no threads. Once a new thread got created from a CPU session, it can be assigned to the task by calling:

```
int bind_thread(Capability thread);
```

3.7 CAP - allocator for capabilities

A capability is a system-wide unique object identity that typically refers to a remote object implemented by a service. For each object to be made remotely accessible, the service creates a new capability associated with the local object. CAP is a service to allocate and free capabilities:

```
Capability alloc(Capability ep_cap);  
void free(Capability cap);
```

The `alloc` function takes an endpoint capability as argument, which is the communication receiver for invocations of the new capability's RPC interface.

3.8 LOG - debug output facility

The LOG service is used by the lowest-level system components such as the `init` process for printing debug output. Each LOG session takes a `label` string as session argument, which is used to prefix the debug output of this session. This enables developers to distinguish multiple producers of debug output. The function

```
size_t write(const char *string);
```

outputs the specified `string` to the debug-output backend of core.

4 Process creation

The previous section presented the services implemented by core. In this section, we show how to combine these basic mechanisms to create and execute a process. Process creation serves as a prime example for our general approach to first provide very simple functional primitives and then solve complex problems using a composition of these primitives. We use slightly simplified pseudo code to illustrate this procedure. The `env()` object refers to the environment of the creating process, which contains its RM session and RAM session.

Obtaining the executable ELF binary

If the binary is available as ROM object, we can access its data by creating a ROM session with the binary's name as argument and attaching its dataspace to our local address space:

```
Capability file_cap = session("ROM", "filename=init");
Capability ds_cap = Rom_session_client(file_cap).dataspace();

void *elf_addr = env()->rm_session()->attach(ds_cap);
```

The variable `elf_addr` now points to the start of the binary data.

ELF binary decoding and creation of the new region map

We create a new region map using the RM service:

```
Capability rm_cap;
rm_cap = session("RM", "");
Rm_session_client rsc(rm_cap);
```

Initially, this region map is empty. The ELF binary contains CODE, DATA, and BSS sections. For each section, we add a dataspace to the region map. For read-only CODE and DATA sections, we attach the corresponding ranges of the original ELF dataspace (`ds_cap`):

```
rsc.attach(ds_cap, size, offset, addr);
```

The `size` and `offset` arguments specify the location of the section within the ELF image. The `addr` argument defines the desired start position at the region map. For each BSS and DATA section, we allocate a read-writeable RAM dataspace

```
Capability rw_cap = env()->ram_session()->alloc(section_size);
```

and assign its initial content (zero for BSS sections, copy of ELF DATA sections).

```
void *sec_addr = env()->rm_session()->attach(rw_cap);
... /* write to buffer at sec_addr */
env()->rm_session()->detach(sec_addr);
```

After iterating through all ELF sections, the region map of the new process is completely initialized.

Creating the first thread

For creating the main thread of the new process, we create a new CPU session from which we allocate the thread:

```
Capability cpu_cap = session("CPU", "");
Cpu_session_client csc(cpu_cap);
Capability thread_cap = csc.create_thread();
Cpu_thread_client ctc(thread_cap);
```

When the thread starts its execution and fetches its first instruction, it will immediately trigger a page fault. Therefore, we need to assign a page-fault handler (pager) to the thread. With resolving subsequent page faults, the pager will populate the address space in which the thread is executed with memory mappings according to a region map:

```
Capability pager_cap = rsc.add_client(thread_cap);
ctc.set_pager(pager_cap);
```

Creating an address space (task)

The new process' protection domain (task) corresponds to a TASK session:

```
Capability task_cap = session("TASK", "");
Task_session_client tsc(task_cap);
```

Assigning the first thread to the task

```
tsc.bind_thread(thread_cap);
```

Starting the execution

Now that we defined the relationship of the process' region map, its main thread, and its address space, we can start the process by specifying the initial instruction pointer and stack pointer as obtained from the ELF binary.

```
ctc.start(ip, sp);
```


5 Framework infrastructure

Apart from the very fundamental mechanisms implemented by core, all higher-level services have to be implemented as part of the process tree on top of core. There are a number of frameworks at hand that provide convenient interfaces to be used by such components. In this section, we outline the most important frameworks.

5.1 Communication

The basic mode of operation of our RPC framework is based on C++ streams. It uses four different stream classes: `Ipstream` for sending messages, `Ipc_istream` for receiving messages, `Ipc_client` for performing RPC calls, and `Ipc_server` for dispatching RPC calls. In the following, we use illustrative examples.

Sending a message

```
Ipstream sender(dst, &snd_buf);
sender << a << b << IPC_SEND;
```

The object `sender` is an output stream that is initialized with a communication endpoint (`dst`) and a message buffer (`snd_buf`). For sending the message, we sequentially insert both arguments into the stream to transform the arguments to a message and finally invoke the IPC mechanism of the kernel by inserting the special object `IPC_SEND`.

Receiving a message

```
int a, b;
Ipc_istream receiver(&rcv_buf);
receiver >> IPC_WAIT >> a >> b;
```

For creating the `receiver` input stream object, we specify a receive message buffer as argument that can hold one incoming message. By extracting the special object `IPC_WAIT` from the receiver, we block for a new message to be stored into `rcv_buf`. After returning from the blocking receive operation, we use the extraction operator to *unmarshal* the message argument by argument.

Performing a RPC call

```
Ipc_client client(dst, &snd_buf, &rcv_buf);
int result;
client << OP_CODE_FUNC1 << 1 << 2
      << IPC_CALL >> result;
```

The first argument is a constant that references one among many server functions. It is followed by the actual server-function arguments. All arguments are marshalled into the `snd_buf`. When inserting the special object `IPC_CALL` into the `client` stream, the client blocks for the result of the RPC. After receiving the result message in `rcv_buf`, the RPC results can be sequentially unmarshalled via the extraction operator. Note that `rcv_buf` and `snd_buf` may use the same backing store as both buffers are used interleaved.

Dispatching a RPC call

```

Ipc_server server(&snd_buf, &rcv_buf);
while (1) {
    int opcode;
    server >> IPC_REPLY_WAIT >> opcode;
    switch (opcode) {
        case OP_CODE_FUNC1:
            {
                int a, b, ret;
                server >> a >> b;
                server << func1(a, b);
                break;
            }
        ..
    }
}

```

The special object `IPC_REPLY_WAIT` replies the request of the previous server-loop iteration with the message stored in `snd_buf` (ignored for the first iteration) and then waits for an incoming RPC request to be received in `rcv_buf`. By convention, the first message argument contains the opcode to identify the server function to handle the request. After extracting the opcode from the `server` stream, we branch into a server-function-specific wrapper that reads the function arguments, calls the actual server function, and inserts the function result into the `server` stream. The result message is to be delivered at the beginning of the next server-loop iteration. The two-stage argument-message parsing (the opcode to select the server function, reading the server-function arguments) is simply done by subsequent extraction operations.

5.2 Server framework

Each component that makes local objects remotely accessible to other components has to provide means to dispatch RPC requests that refer to different objects. This procedure highly depends on the mechanisms provided by the underlying kernel. The primary motivation of the server framework is to hide actual kernel paradigms for communication, control flow, and the implementation of local names (capabilities) behind a generic interface. The server framework unifies the control flow of RPC dispatching and the mapping between capabilities and local objects using the classes depicted in Figure 5.

Object_pool is an associative array that maps capabilities from/to local objects. Because capabilities are protected kernel objects, the object pool's functionality is supported by the kernel.

On L4v2 and Linux, capabilities are not protected by the kernel but are implemented as unique IDs. On these base platforms, the object pool performs the simple mapping of such unique IDs to object pointers in the local address space.

Server_object is an object-pool entry that contains a dispatch function. To make a local object type available to remote components, the local object type must inherit from `Server_object` and provide the implementation of the dispatch function as described in Section 5.1.

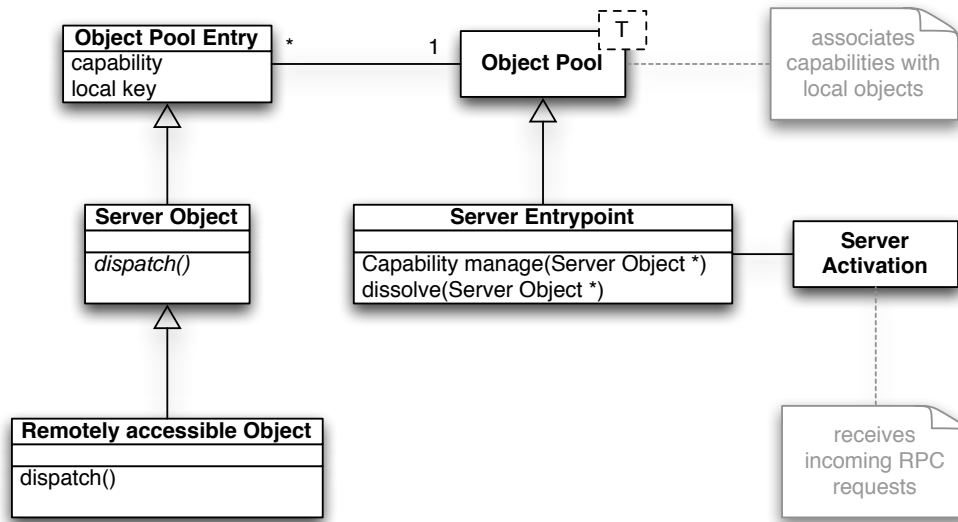


Figure 5: Relationships between the classes of the server object framework

Server_entrypoint is an object pool that acts as a logical communication entrypoint. It can manage any number of server objects. When registering a server object to be managed by a server entrypoint (`manage` method), a capability for this object gets created. This capability can be communicated to other processes, which can then use the server object’s RPC interface.

Server_activation is the stack (or thread) to be used for handling RPC requests of an entrypoint. The interface of the server entrypoint envisions to attach multiple server activations to one and the same server entrypoint to enable the concurrent handling of multiple RPC requests.

On L4v2 and Linux however, exactly one server activation must be attached to a server entrypoint. This implicates that RPC requests are handled in a strictly serialized manner and one blocking server function delays all other pending RPC requests referring the same server entrypoint. Concurrent handling of RPC requests should be realized with multiple (completely independent) server entrypoints.

5.3 Process environment

As described in Section 2, a newly created process can only communicate to its immediate parent via its parent capability. This parent capability gets created “magically” dependent on the actual platform.

For example, on the L4v2 platform, the parent writes the information about the parent capability to a defined position of the new process’ address space after decoding the ELF image. On the Linux platform, the parent uses environment variables to communicate the parent capability to the child.

Before entering the `main` function of the new process, the process’ startup code `cr0` is executed and initializes the *environment* framework. The environment contains RPC communication stubs for communicating with the parent and the process’ RM session, CPU session, TASK session, and RAM session. Furthermore, the environment contains a heap that uses the process’ RAM session as backing store. The environment can be used from the actual program by dereferencing the pointer returned by the global function:

```
Env *env();
```

5.4 Child management

The class `Child` provides a generic and extensible abstraction to unify the creation of child processes, serve parent-interface requests, and to perform the book keeping of open sessions. Different access-control and resource-trading policies can be realized by inheriting from this class and supplementing suitable parent-interface server functions.

A child process can be created by instantiating a `Child` object:

```
Child(const char *name,
      Capability elf_ds_cap,
      Capability ram_session_cap,
      Capability cpu_session_cap,
      Cap_session *cap_session,
      char *args[])
```

The name parameter is only used for debugging. The args parameter is not yet supported.

5.5 Heap partitioning

In Section 1 where we introduced the different types of components composing our system, we highlighted *resource multiplexers* as being critical for maintaining the isolation and independence of applications from each other. If a flawed resource multiplexer serves multiple clients at a time, information may leak from one client to another (corrupting isolation) or different clients may interfere in sharing limited physical resources (corrupting independence). One particular limited resource that is typically shared among all clients is the heap of the server. If the server performs heap allocations on behalf of one client, this resource may exhaust and renders the service unavailable to all other clients (denial of service). The resource-trading concept as presented in Section 2.2 enables clients to donate memory quota to a server during the use of a session. If the server's parent closes the session on request of the client, the donated resources must be released by the server. In order to comply with the request to avoid intervention by its parent, the server must store the state of each session on dedicated dataspace that can be released independently from other sessions. Instead of using one heap to hold anonymous memory allocations, the server creates a *heap partition* for each client and performs client-specific allocations exclusively on the corresponding heap partition. There exist two different classes to assist developers in partitioning the heap:

Heap is an allocator that allocates chunks of memory as dataspace from a RAM session. Each chunk may hold multiple allocations. This kind of heap corresponds loosely to a classical heap and can be used to allocate a high number of small memory objects. The used backing store gets released on the destruction of the heap.

Sliced_heap is an allocator that uses a dedicated dataspace for each allocation. Therefore, each allocated block can be released independently from all other allocations.

The `Sliced_heap` must be used to obtain the actual session objects and store them in independent dataspace. Dynamic memory allocations during the lifetime of a session must be performed by a `Heap` as member of the session object. When closing a session, the session object including the heap partition gets destroyed and all backing-store dataspace can be released without interfering other clients.

6 Limitations and Outlook

In its current incarnation, the design is subject to a number of limitations. As a prime example for managing resources, we focused our work on physical memory and ignored other prominent resource types such as processing time, bus bandwidth, and network bandwidth. We intend to apply the same methodology that we developed for physical memory to other resource types analogously in later design revisions. We do not cover features such as virtual-memory or transparent copy-on-write support, which we regard as non-essential at the current stage. At this point, we also do not provide specifics about the device-driver infrastructure and legacy-software containers. Note that the presented design does not fundamentally contradict to the support of these features. To keep the design space at a manageable dimension, we have *purposely* excluded these items from the initial set of problems to address.