Advanced Operating System
Report Group D

Christian Fenia, Philipp Schaad, Jan Müller

December 22, 2017

Contents

Chapter 1

ClownFish

Throughout this project we have tried to keep most of our individual subsystems and libraries as simple as possible. This is to support the mentality that our code should be easy to understand (well ok that didn't always work) and easy to debug and reason about. Keeping things like data structures simple, easy to understand, and general purpose sometimes comes at a cost of performance. We have decided that potential performance gains achieved by over-complicating our system and book-keeping mechanisms would be negligible compared to the additional cost in complexity.

1.1 Memory Management

Our memory manager (`libmm`) is kept relatively simple, such that reasoning about it can be done with ease. In essence each core in our system takes care of memory management by itself, receiving exactly half (in a two core system, less otherwise) of the total physical system memory for itself at boot. Each processor core then has an instance of `libmm` running and manages distribution and tracking of its own segment of physical memory.

1.1.1 Resource Tracking

To track what system resources are free or in use, we employ a very basic doubly linked list, where each element (called a node) has a type, a capability for the memory region it represents, as well as pointers to the next and previous node together with its region's base address and size in bytes.

We track three distinct types of memory regions with our nodes; free, allocated, and wasted. Free and allocated are fairly self explanatory. The 'wasted' region type is rather rare, but gets used as a workaround for our incomplete capability system. We will go into detail on this in a little bit, but essentially it's a failsafe for when we try to re-allocate a previously used region of memory, since we were unable to correctly revoke that region's capability.

Initially our list only contains one node, which represents the entire physical memory managed by this instance of `libmm`. Upon allocation we split this region up, forking off a region of the size that was requested and shrinking the free node accordingly.

To look for a free region of memory we have implemented a naive algorithm which simply traverses the doubly linked list starting at the head, until a node is reached that has at least the size of the region we are looking for and is marked as free. This gives us a worst case complexity of $O(n)$ when allocating. We can reach this worst case by continuously allocating (rather small) regions of memory without freeing, which implies that our list keeps growing as we fork off new allocated regions from our one contiguous free region at the tail, and we will always have to scan $m + 1$ regions after m allocations.

We decided to use this approach because it is easiest to implement and understand, therefore being less error prone and easier to maintain and upgrade. In addition to that the performance does not get impaired in the standard case, thus making this a viable option. A switch to a binary tree did not seem attractive enough to be worth the additional book-keeping and structure management, which in itself would cause overhead. We have also decided to not perform rearrangement of allocated memory to reduce fragmentation, as that would vastly increase the book-keeping complexity.

1.1.2 Capability Handling

As mentioned in subsection 1.1.1 we initially hold one big node that represents all of our free memory. We section up our device memory into two equally sized pieces in the `'menu.lst.armv7_omap44xx'`, and upon booting each processor's `libmm` gets one of those regions. We represent this region by creating one big ram capability and assigning it to our initial node. This ram capability never changes. Instead we always retype this initial capability, where the new capability represents the offset of the newly allocated region to our original ram capability.

This approach gives us three major advantages: Firstly, the management and tracking of capabilities is easier. If we free the memory backed by one of those retyped capabilities we can simply destroy its representing capability. In addition to that, if we allocate memory we now do not have to worry about the chance that there might already be a retyped capability for this section of memory. Secondly, destroying the memory manager remains a simple task this way, leaving us solely with the responsibility of destroying all capabilities, instead of having to revoke each one of the descendants. It

```
addr allocate(size) {
    ROUND_UP(size, BASE_PAGE_SIZE);
    Node *node = find_node(size);
    if (node->size > size) {
        node->size -= size;
        node = create_and_insert_new_node(size);
    }
    node->type = ALLOCATED;
    err = cap_retype(node->cap, initial_ram_cap, offset, size);
    if (err_is_fail(err)) {
        node->type = WASTED;
        return allocate(size);
    }
    return node->addr;
}
```

Listing 1.1: Allocation Procedure (Pseudocode)

would be harder to keep track of all of those. And finally, with this approach it is easy to add new ram to the manager should new one appear for some reason. All we need to do in that case is to check whether the current big initial capability has run out of free memory, and use the new one instead.

1.1.3 Allocating memory

As mentioned before, our allocation procedure is rather simplistic. We simply start to traverse the list of all memory regions, until we find a match where the requested size fits (the region is greater than or equal to the requested size), and where the region is marked with type free. Note that the size we request to allocate is adjusted to be a multiple of the base page size. If we have found a node that has exactly the size we requested, we simply change its type to allocated. If the node is bigger, we shrink it and add a new node with the size we wanted before it, inserting both back into the list. In both cases we update the capability of our freshly allocated node by retyping the initial ram capability with the correct size and offset.

This is where the node-type `'wasted'` comes into play. Since we cannot correctly revoke capabilities, it can happen that this retype operation fails if the node has previously been allocated and freed. In that case we simply set the type to wasted and recursively re-start the allocation procedure to recover. Listing 1.1 shows a pseudo-code outline of this procedure.

1.1.4 Freeing memory

Freeing memory is quite simple. We revoke and destroy the capability (if it still exists). We set the indicator to free and check if there is a free node right before or after the just freed node. If so, we merge the free spaces together so that we can allocate bigger contiguous spaces of memory.

1.1.5 Slab refills

To refill the slabs, we need memory. To allocate more memory we need slabs. To make sure that we don't run into a deadlock, we made sure that the slab gets refilled when we have less than a few slabs left.

1.2 Memory Mapping

1.2.1 Structures and Bookkeeping

To keep track of the various mapped regions and pieces of memory, as well as the free virtual addresses, we have to keep a significant amount of state for the memory mapping and paging process.

- Free vspace: To prevent address collisions, the free virtual addresses are tracked. This is done using a singly linked list, because its simplicity. This might introduce additional overhead when mapping memory, when the process has run some time and the memory is fragmented. The lookup scales with $0\,(n)$. Currently, freed nodes are simply prepended to the list. This scales with $0\,(1)$
- Mappings: To be able to unmap a piece of memory again, the mappings are stored. This is done using a singly linked list. A new mapping is added in front of the list. This scales with $0\,(1)$. Finding the right mapping for unmapping memory scales with $0\,(n)$ with the number of mappings in the worst case, when it has to traverse the whole list. However, we think that there are two sorts of mappings. One is short lived and will be unmapped soon after it is mapped. And the others are long living. Our implementation let the long living mappings wander to the back of the list while being fast for unmapping recently added mappings.
- L1 pagetable: a reference to the L1 pagetable is stored.
- L2 pagetables: an array of L2 pagetables. Initially, this list is empty. If a L2 pagetable is used for the first time, it is created and the capref is stored for future uses.
- Spawninfo: When spawning a new domain, we need to copy the caps for the mappings and the L2 pagetables to the new domain. To be able to reuse our mapping code, we keep a reference to the spawninfo that contains a callback function to be used when mapping for a foreign domain.

1.2.2 Mapping

The mapping of a memory frame to the virtual address space of a domain consists of the following steps:

1. If the address is not user chosen, the information about a free block of virtual addresses is computed from with the information stored in the paging state (see subsection 1.2.1).
2. The L2 pagetable corresponding to the virtual address to be mapped is read from the L1 pagetable. If this pagetable does not yet exist, a new L2 pagetable is created.
3. Map the minimum between the number of bytes we have to map and the number of bytes that still fit into the L2 pagetable.
4. Store the reference to the L2 pagetable and the mapping information to be able to unmap the piece of memory again.
5. The steps 2 - 4 are repeated until all memory is mapped. This is the case when the requested size, starting from the virtual address, does not fit into a single L2 pagetable.

1.2.3 Unmapping

Because we stored a fair amount of state, unmapping is easy. All parts of the region to unmap are traversed and unmapped. After this is done, the freed virtual addresses are added to the list of free vspace again (see subsection 1.2.1).

One problem we encountered while implementing the unmapping was that it can be hard to test or demonstrate. Due to compiler optimizations (especially instruction reordering), unmapped memory seemed accessible even after it was unmapped for a short time.

1.3 Demand Paging

1.3.1 Pagefault handling

A pagefault is triggered when virtual memory is accessed, that has not been backed with physical memory. Before handling this exception, we do some sanity checks on the accessed address. We check whether the caller tries to dereference a NULL pointer, if it is trying to access something in the kernel space (above 0x80000000) or if the current stack pointer exceeds the stack. Such a behaviour means misbehaviour or a programming error and the calling thread is killed.

After these checks, the pagefault is handled. We decided to use BASE_PAGE_SIZE (4kB) sized pieces of memory to benefit from address locality and reduce the number of pagefaults. First, the memory is requested from the init domain using a non-malloc path (see subsection 1.3.2). After we received the memory, it is mapped to the caller's virtual address space (see section 1.2).

We did a quick evaluation of other sizes of memory. We run the init domain and let it run our test suite and create 5 threads. This resulted in a total of 18 pagefaults (since the start of the demand paging mechanism). We repeated the test but increased the amount of memory that is mapped by a factor of 2. This resulted in 12 pagefaults, which means a reduction of 33%. Because the overhead of our pagefault handling is no bottleneck to the system and the savings are therefore not that big, we did not implement the larger memory size. The increase of the size would generate additional complexity on the programming side, as we would have to check first, if the next page is already mapped and would need additional testing to make sure that we cover all corner cases.

The pagefault handler operates on a separate interrupt stack. We encountered an error, in which the handling of the pagefault would exceed the initial size of 4kB. Increasing the size of the stack to a really large value (16kB) solved that problem.

1.3.2 RAM alloc in pagefault handling

Our initial pagefault handler occasionally caused pagefaults while handling pagefaults, which led to system crashes. We circumvented this behaviour by making sure, that the pagefault handler receives a fully stack based (no memory allocation) path for requesting a new ram capability. The following additional steps were taken, to allow an as direct handling of the pagefault as possible:

- Prevent scheduling of other threads during pagefault handling
- Create a separate lmp_channel for ram cap requests during pagefaults
- Create a separate waitset for the use with the new lmp_channel, to make sure that we give control to the domain providing ram capabilities. (And back, of course)

For more information see section 1.4.3.

1.4 RPC

For terminology, our actual remote procedure call system is split into two sections: The RPC system, which is build on lmp_chan's and for transferring messages between processes on the same core, and the URPC system, which is build on a shared memory page and is for transferring messages between the init processes on the two cores.

Now, our biggest regret is that we never unified those two systems properly. An approach at unification which would have involved rewriting aos_rpc to account for that could have likely provided an uniform interface and saved us a lot of pain along the way. But we didn't get there.

1.4.1 Early struggles

The development of the RPC system started out as an adhoc implementation of just adding one RPC call after another and having all the logic in every single handler. This was, naturally, horribly error prone and hard to maintain or upgrade. Also, it was based on a misunderstanding on how to approach this. We registered a send and recv handler on the channel every time we made one of those RPC calls, this turned out to lead to massive issues with multithreading and meant processes could only receive messages if they had just sent one. We then had a single shared recv handler, but the memory RPC call needed its own recv handler as that design didn't quite work for that one. "The memory RPC call needs to be handled extra" is a common theme that would repeat throughout our changes to this system.

At first we tried to make some macros and functions while keeping the same basic structure, but that too turned out to be unmanageable. The time for a rewrite of the first version of the current system had come.

1.4.2 Rewrite

Instead of init and other processes both doing their own ad-hoc things with the RPC setup and calls, we developed a single shared framework. This framework is set up in such a way that first we automatically grab a fresh ID to mark the current logical call. This way we can associate responses with it. After that, the memory for the logical call gets persisted and enqueued in the list of messages to send. Further, we register the message on the channel for sending if nothing is registered there yet. When it is time to send, the message automatically gets a standardised encoding scheme applied to it: the first 32bit word of every physical message gets encoded to contain the type of call, id of the call and length of the total payload of the logical message this is part of. Then, the actual sending takes place until all parts of the message are sent or an issue occured.

Further, the framework also has a standardised receive handler, which receives all calls on that channel and recovers the logical message from the physical messages sent. It does so based on the type, id and a list of unfinished calls to match the type and id to. When a message is fully reassembled, we call a process/channel specific handler. Here, an optimisation opportunity for small messages (which are the vast majority) presented itself. Instead of persisting it all and adding it to the list and doing all the other steps involved, we can do it malloc-free by passing that element to the handler on the stack. This works because we return into this function again and in the normal case free the memory.

While this framework now standardised the encoding of messages and made it really convenient to work with rpc calls, it also presented a large increase in complexity of the code in one particular place. However, it reduced the overall complexity by reducing the complexity and gain managability in many other places. The rewrite, however, still had issues.

1.4.3 RAM rpc calls

After the first series of minor bugs was fixed, we ran into a pretty big issue: The ram RPC handler now ran over that system, which used mallocs in various places due to its generality and the thus arising need to persist various sizes etc. Our first and naive approach was to add special conditions for ram_rpc calls. We relied on using buffers of the structs involved instead of allocating new memory and therefore taking a slightly special path. This, while working for some time, was naturally doomed to fail as we still got pagefaults at times and the pile of patches that held our os together was ever increasing.

The next step then was to move the ram_rpc procedure partially out of the system. Receiving calls is still part of it, sending is not. This cut down on the amount of memory allocations further and we had a buffer for the rest, which got us a bit further. But, the actual solution to this problem came later, after it had plagued us for weeks with occurring only spuriously and the code around it being annoyingly complex and hard to comprehend. The solution was, that the ram RPC got moved into its own channel, independently of the others. Up to this point, all domains created only one single channel to init. We gave the ram rpc call its own machinery, very reminiscent of the original implementation. Luckily, on the init side, it did not have to be handled differently. However, we still got pagefaults and the occasional strange errors, which later got tracked down to a critical bug in the sending routine of the RPC system. It used only a single queue for sending instead of a per-channel queue, which meant that in particular cases it was possible for messages not to get sent to the proper channel. In other cases, a message was tried to send despite the wrong channel being ready. This got fixed after a lot of painful investigation as it was hard to track down that this caused the errors.

```
errval_t init_rpc_server(
            void (*recv_deal_with_msg)(struct recv_list *),
            struct lmp_chan *chan);

errval_t init_rpc_client(
            void (*recv_deal_with_msg)(struct recv_list *),
            struct lmp_chan *chan,
            struct capref dest);
```

Listing 1.2: RPC init function prototypes

```
errval_t send(struct lmp_chan *chan, struct capref cap,
             unsigned char type, size_t payloadsize,
             uintptr_t *payload,
             struct event_closure callback_when_done,
             unsigned char id);
```

Listing 1.3: RPC send function prototype

1.4.4 Implementation

The current RPC subsystem system is split in two parts. In aos_rpc_shared is (most of) the machinery described so far.

The most important functions are shown in Listing 2.1. init_rpc_client creates a new lmp_chan. It creates it with an empty remote_cap, waiting for anyone to connect to it. Also, it uses the handler provided as recv_deal_with_msg as the callback to use when a logical message is fully reconstructed on this channel. The struct recv_list contains the payload (both the data and the eventual capref) and its metadata (rpc call type, id, size). This is logically the same as init_rpc_server, except there you need to provide a remote cap to connect to.

Listing 1.3 depicts the primary way to send messages, it does all the explained converting, enqueuing and registering (send_loop does the actual partitioning and sending).

Something that at the time seemed useful was having a callback to be called once the message was fully sent, but in practice our system never evolved to be callback based enough for this to become useful. A feature missing, is a callback to be called in case sending fails, so that we can do custom and message specific handling of issues.

Finally, Listing 1.4 shows the convenience function send_response, which answers a RPC call with the given cap and data. The real gain, however, was not in having those functions, but that they allowed us to write the rpc_framework function (shown in Listing 1.5)), which looks really wild - and it is. It works in conjunction with a logical message recv handler which

```
errval_t send_response(struct recv_list *rl,
                       struct lmp_chan *chan,
                       struct capref cap,
                       size_t payloadsize,
                       void *payload);
```

Listing 1.4: RPC send_response prototype

```
rpc_framework(void (*inst_recv_handling)(void *arg1,
                  struct recv_list *data),
              void *recv_handling_arg1,
              unsigned char type,
              struct lmp_chan *chan,
              struct capref cap,
              size_t payloadsize,
              uintptr_t *payload,
              struct event_closure
                  callback_when_done)
```

Listing 1.5: rpc_framework prototype

is set up in such a way that it takes the type and id info we were given and looks it up in the list of calls that we made and which we are await an answer from.

rpc_framework takes the function which handles the response to the call are about to make, as well as some state for it (arg1 will be passed to the handler). The other parameters are used to provide the info needed for making the call. Additionally, it registers for a response. After that, it sends out the message and pumps the waitset until it receives an answer (thus being blocking). This means all we now needed to do to add another RPC call is to implement a very simple recv handler and serialise our data for sending.

Listing 1.6 shows an example for a call for getting the nameserver's cap (so we can connect with all the other services by asking the nameserver for those).

Overall, the RPC system had a long history and features a great API now. One that is simple to use and does not need a deep understanding of how lmp_chan's work. However, this was paid for in bugs, complexity and a lot of time. However, in the end, it was a worthwhile tradeoff as this setup can be extend rapidly.

However, our one big drawback is that we never made it backend agnostic to integrate the cross-core communication system (dubbed URPC) into it.

```
static void get_nameserver_recv_handler(
    void *arg1, struct recv_list *data)
{
    struct capref *retcap = (struct capref *) arg1;
    *retcap = data->cap;
}

errval_t aos_rpc_get_nameserver(
    struct aos_rpc *rpc, struct capref *retcap)
{
    rpc_framework(get_nameserver_recv_handler,
                  (void *) retcap,
                  RPC_TYPE_GET_NAME_SERVER,
                  &rpc->chan, NULL_CAP,
                  0, NULL, NULL_EVENT_CLOSURE);
    return SYS_ERR_OK;
}
```

Listing 1.6: aos_rpc_get with helper function

1.5 URPC

As we do not have a fully unified system, we have the init<->init cross core communication system which is dubbed URPC.

Originally it was just a single page split into two 2 roughly 2kb section (slightly less as we had a smaller shared head at the top of the page) and could just transfer messages in 2kb blogs and signalling the other side that the message arrived.

This got adapted when the actual cross core communication chapter came half a week later and required a handling using cache lines. So we adapted it to be chunked and used that opportunity to adopt it to a similar framework system as the RPC one (see 1.4.4 for more details).

As the URPC system does not involve the kernel, it turned out to be a much simpler system and way less error prone. We segmented the page into 64byte chunks. 1 byte is used for flags and 63 bytes are used for data. We kept track of these chunks using a ringbuffer. We learned from the RPC system here and did a better encoding. Because urpc only has to deal with a single "channel" and therefore a single FIFO queue of messages, we only send the length and the metadata once per logical message. The flag byte is primarily used to signify if this entry is empty or written to, as well as what encoding we will be using.

A good side-effect of our implementation is, that because we send and receive in a loop, we only need a single memory barrier in each loop.

On top of these two core loops, we run an endless loop which polls for available messages or messages ready to be sent.

On top of that is a somewhat familiar machinery. We can enqueue messages and register receive handlers.

For implementing the actual URPC calls we first considered making an RPC equivalent for every single URPC call, but then instead added some more logic which allowed us to root an arbitrary RPC call to init over URPC, thus allowing the two init processes to act as if they were a single one for the purposes of RPC. An init process receives a fake call to its rpc receive handler with a normal recv_list struct pointer, except that the channel points to NULL, which can never happen in a normal RPC call and is thus a fairly obvious and simple distinguisher between an RPC or URPC call.

In theory, this same system could be used to allow an arbitrary pair of processes to communicate directly with each other, if they can be arranged to share memory. This, however, we never got around to implementing and instead just implemented a proxy service using the init processes for communication. This has the benefit of not having the extra looping thread overhead and is easier to set up than sharing physical memory across cores (since that would require us to mint and track a new ram cap. Our current memory manager and the incomplete implementation of cap_revoke would render this difficult.)

1.6 Spawning Processes

Our spawning code is nothing fancy and pretty much follows the procedure outlined by the book. Let's go over the individual steps.

Since we do not have a file-system in our OS, we are dealt a lucky hand in this portion. All we have to do to find the correct module is look up the name in the multiboot image. Since the call to `spawn_load_by_name` of our spawn library can (but does not have to) contain an arbitrary number of arguments, the first thing we do is parse the requested name, filtering out the actual program name and separating the arguments away. We now look up that name in the multiboot image. Once we have found our module, we identify the frame it sits in and map it into our own address space with `paging_map_frame`, which gives us a virtual address pointing to our ELF file.

In the next step we need to set up a bunch of capabilities for our new process. The first one would be an L1 CNode, with and into which we then create a bunch of L2 CNodes using `cnode_create_foreign_l2` (since we have already created the L1 node). We map the TASK CNode (which contains information about the process) into the child process's capability space, and then back all it's L2 CNodes with some (exactly a page) of memory.

We now have to set up the child process's virtual address space by creating an L1 page-table in our current virtual address space, which we then copy in to the child process's L1-page-table slot. This step is followed by calling `paging_init_state`, which can now set up the paging state of our new process.

We are now ready to load the ELF binary and initialize the dispatcher that will take care of launching our process. This is a rather tedious process of just creating the dispatcher and an endpoint to it and then copying everything into the child process's virtual address space. In there, we then fill in the dispatcher struct with details like where the GOT sits, what the process's domain ID will be initially and what core it will be run on.

All that remains to be done now before we invoke the dispatcher, which will start our process, is to set its arguments correctly. In this step we will check if we have previously received any arguments from the user. If so, we will map those into the child process, if not, we will look up if the multiboot image provides some arguments and use those instead. After this we are done and are ready to run our process by invoking the dispatcher.

Chapter 2

Individual Projects

2.1 TurtleBack (Shell) (Philipp Schaad)

The shell is a very essential part of any OS, as it allows the user to interact with the system at its most basic level. However, for the shell to make any sense to begin with, one first needs a way to talk with the user. This is typically done with a standard text interface. Our system thus needs to have the ability of printing text to the screen, and read text back that has been typed in by the user. It is best to do this reading and writing as centralized as possible, and for this we will first look at the subsystem responsible for that bidirectional input and output; the terminal driver.

2.1.1 Terminal Driver

Our terminal driver is kept as simple and protocol free as possible. For this reason it is running in the `init` domain on both cores. Let's first discuss what this implies for our output to the screen.

When printing to the screen with functions from the C standard library, the system first checks whether we are in the `init` domain or not. If we are indeed, we directly talk to the terminal driver, requesting to have it print to the screen for us. The terminal driver (since it is inside of the `init` domain) can do so via system call. If we are however in a different domain, we will send out an RPC to `init`, which in turn uses the terminal driver to service our request and print to the console via system call. This can happen on both cores simultaneously and entirely independently, which keeps the system's complexity at a minimum.

Input is a little less trivial, but still very simple. Since we have two instances of our terminal driver running (one on each core), we will call the one on core 0 our master and the one on core 1 our slave instance. Since we can only have at most one instance registered for interrupts from the UART interface, we will let the master instance take care of interrupt handling, and hence register it to receive UART interrupts. If an interrupt is received by the master instance, it will perform a system call to retrieve the typed in character from the UART buffer and store it in its own input buffer (which we will get to in a second). Since the slave instance will not receive that interrupt and does not have the ability to retrieve that same character from the UART again, the master instance performs a URPC with which it sends the received character to the slave instance, which can add it to its own buffer.

2.1.1.1 Input Buffers

Our terminal driver supports two separate modes of buffering input from the UART interface. Line- and direct-mode.

In line-mode we buffer - as the name suggests - entire lines of text. Our line buffer consists of two separate character arrays, one of which is considered to be our write-buffer and the other one is the read-buffer. When a new character arrives (via interrupt if we are on core 0, via URPC otherwise), we will first perform a little bit of preprocessing with it. The most obvious step is to check if it is a character or signal indicating the end of a line (EOT, LF, CR). In that case we null terminate our write-buffer and switch the two buffers. So now when someone tries to read from the read-buffer, it will get a character from the newly finished line, and we will start writing to a new (now invalid and thus considered to be empty) line. The remaining preprocessing steps are less important and mainly matter for the second point where the two modes differ from each other; echoing. In line-mode let the terminal driver manage echoing of newly arriving characters, and not whatever process is reading that input. This is also why we do not use this mode for the shell, but instead use our second mode of operation.

In direct-mode the steps performed by the terminal driver are kept to a bare minimum. Our buffer has the form of a ring-buffer (FIFO), and whenever a new character arrives, we simply add it to the buffer and advance the tail by one, without really caring about what the character value was. This gives the mode its name, as we will directly push ALL input forward to the consuming process(es), without performing any cleanup or echoing the characters back to the user. Our consumer has to take care of that.

In both of those modes we simply discard new characters if our buffer is already full. In addition to that, escape and control sequences that make a mess or might be important for the consumer are analyzed by the terminal driver and stored as one special character (instead of their raw form, which typically is 2-3 characters). This can be easily done, since the ASCII standard for our supported symbols only consumes 128 values. A character (byte) can hold 256 values, so we can use the remaining spaces to encode 128 custom characters. For example left-, up-, right-, and down-arrow keys will get escaped by the terminal driver and stored as special values, so the consuming process understands that they were pressed.

When a process performs an input operation using one of the C standard library functions, we differentiate between `init` domain and other domains again. In the case of `init`, we directly query the terminal driver for new characters. Other domains will perform an RPC to init, which will serve the character via the terminal driver. In both cases, the call to the terminal driver will block for as long as the input-buffer does not have any ready characters. As soon as characters arrive, one is returned and the consumer process can analyze it. Which input-buffer will get queried depends on the currently selected mode in the terminal driver. To make sure the two driver instances' input-buffers are in sync, whenever a character is removed from one of the two instances, it will fire a URPC to the other instance, moving the head by one space, thus also removing the character.

2.1.2 The Shell

Our shell is a separate process (`turtleback`) that can be spawned on either core once the system has completed its boot process (core 0 by default). The main shell process in itself is not very interesting. All its doing is performing an endless loop of reading characters from the terminal (via C standard library functions, which go over `init`'s terminal driver), storing them in a buffer and then parsing that line once the user hits a new line. The interesting part lies in just the parsing of that input line and what that sets in motion.

The shell serves two main functionalities. It presents the user with a set of built in functions - with the unspectacular name 'builtins' - that can be used to interact with the operating system, and it will spawn other processes for the user when he types their name into the console. Let us first examine the built in functions that our shell provides.

2.1.2.1 Builtins

- `help` - This is probably the most basic builtin that any shell (or application for that matter) should have. It does exactly what the name says; it prints out a list of all available builtin functions and offer a helpful little explanation of what each of them does. Supplying it with the name of another builtin will print more information about how to use said function.

- `clear` - Another relatively self explanatory function, found in pretty much every command-line based interface; it will clear the screen for you (preserving the scroll buffer). This is done by printing the ANSI sequence ESC - [- 2J to the terminal.

- `echo` - Also one of the functions everyone knows; this will simply print whatever arguments you supply it with back out onto a new line in the terminal. It will print all 'words' (arguments) spaced out with one single space, no matter how far apart the arguments were supplied (yes, it behaves exactly like bash's `echo`).

- `ps` - Like its bash equivalent, `ps` will simply perform an RPC to the process manager (in the init domain), which in turn will print a list of all running processes, together with their process ID and which core they are running on.

- `time` - This is the last one of the commands that behaves pretty much like bash; `time` must be used as a prefix command to any other builtin or binary (with arguments or not). It simply times how long it takes for the execution of said command/program to complete. This is done by getting the clock-cycle count before and after the execution, calculating the difference and dividing it by the CPU clock frequency, which gives us the exact time in seconds and milliseconds.

- `led` - A new, but rather unspectacular command, that simply tells the `init` domain to toggle the D2 LED via RPC. If the LED was on it gets turned off, and vice versa. Toggle. That's all. Yes.

- `testsuite` - This performs a little RPC to the `init` domain, which will run a standard test suite that is built in to the OS and contains a couple of memory and regression tests that were used during the development of this operating system. It only exists because we can.

- `memtest` - The memory test will take a single argument (a positive number), that tells it how many bytes the user wants to test. It will then allocate a contiguous region of memory of that size and test it for read- and write-errors.

- `threads` - This is simply a demo of user level threads. It takes a single positive integer argument and will spawn a number of threads equal to that integer argument. Those threads simply print out the line `"Hello world from thread X"`, where `X` is its thread-id.

- `oncore` - Serving as another one of those prefix commands (like `time`), `oncore` will take the program spawning command it prefixes and spawn said binary on the core specified. So if we type in `'oncore 1 hello args..'`, this will spawn the binary `'hello'` on core 1 with arguments `'args..'`. This is done with a standard spawn-process RPC to init, which can already handle cross-core spawns.

- `detached` - This is the final prefix command like `time`. Like `oncore`, it serves as a prefix purely for spawning programs, not shell builtins. Its purpose is pretty simple; `'detached progname args..'` spawns the program `'progname'` with arguments `'args..'`, but will not wait for it to finish before returning to a new shell prompt and listening for user input.

2.1.2.2 Parsing User Input

We parse the user's input every time he or she hits the end of a line and starts a new one (typically by pressing `Ctrl + J`, `Ctrl + D`, or `Return`). The line of text in the buffer gets split up into a series of token. Each space in the line represents the end of a token and the start of the next one, whereby multiple consecutive spaces get dropped and will count as a single token separator. We now look at the first token and treat it as our command, thus deciding what to do based on its value.

We first check if the user wants to execute one of the builtins TurtleBack provides. Our shell holds an internal list of all available built-in functions, together with a handler function attached to each one of them. We can thus simply iterate over this (relatively short) list, performing string matching against each of the identifying commands. If we have a positive match, we will execute that match's handler function, passing all but the first token along as its arguments. If we hit the end of the list without a match, we move on, treating the entered command as a binary the user is trying to spawn.

This leads us to how we handle process spawning. We use an UNIX-like approach, where a program can be launched simply by typing the name of its binary into the console, without a special run command. Internally this will launch an RPC to the `init` process, where we look up if the queried binary name can be found in our multiboot image (since we do not support a file-system, that's the only place we can spawn from). `init` then launches the process (again, passing all the remaining tokens along as arguments to the new program), registering it in our process manager, and returns the PID it just assigned to the process. TurtleBack then queries `init` (via RPC) for the existence of that process, blocking until that PID could not be found anymore. This ensures that we await the termination of our spawned program before showing a new shell prompt and accepting user input, thus allowing the spawned process to take control.

If the requested binary could not be found, the PID returned will have a special value (UINT32_MAX), which tells TurtleBack that we still have not been able to handle the user's command. At this point we know that if we were unable to serve the user's request, he or she must have entered a wrong command, so we echo the command back (just the first token) and tell the user that this does not match to any supported command.

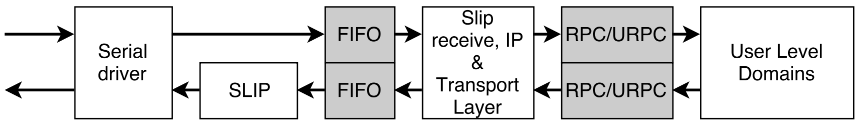


Figure 2.1: The way of a packet from entering on the serial line until leaving the system again. White boxes correspond to individual domains or threads. Gray boxes are used to illustrate the communication channels.

2.2 Network (Jan Müller)

2.2.1 Overview

The implemented network stack is organised on a code level according to the OSI reference layer. However, on the running system, only the transport layer is running isolated from the others. An overview over the implementation can be seen in figure 2.1. The receiving side of the slip protocol, the IP protocol and the transport protocols (UDP, ICMP) are running on the same thread to reduce the overhead introduced by scheduling or the even larger overhead of a call to the RPC system. Another advantage of this system is, that there is less possibility of an internal buffer overflow.
For the communication between different network related domains, an internal network messaging protocol (INMP) was implemented. This protocol relies on the underlying RPC/URPC implementation. This allows for multiple user level domains accessing the network, for example the implemented udp_echo and udp_terminal domains. There is one master domain called network, which starts the network logic. This process takes the own IP address as input. Currently, it is not possible, to change the IP address after this initial setup.
With the current implementation, there are two drawbacks: Because the nameserver is used to connect the network domains and does not support cross core lookups, all network tasks have to run on same core. The other drawback is, that it is not possible to have multiple IP addresses. If one tries to run a second network stack on the other core, the behaviour is not defined as the interrupts are not guaranteed to work as intended anymore.

2.2.2 SLIP

2.2.2.1 Receiving

The serial input is fed into a circular buffer. A thread implementing the SLIP protocol reads from this buffer byte by byte and does the necessary substitutions given by the protocol. Once the message is complete, it is written into a single packet sized buffer. This might lead to problems if the packets are incoming at a higher rate than they are processed. This behaviour was only observed, when the os was hammered by the "flood ping" option of the Linux

ping utility. When sending 500 ICMP echo requests in fast succession, we experienced a 34% packet loss. Increasing the input buffer could postpone this problem to a larger number of incoming packets but would not solve this issue, as messages are still entering the system faster as they are leaving.
In general, the processing should be fast, because it consists only in local comparisons and the reassembly of packet headers for incoming ICMP echo requests. UDP packets are immediately sent to the handler of the corresponding port. A potential overload would show itself in a overflow of the circular buffer which would cause packets to be dropped because of a wrong ip checksum.

2.2.2.2 Sending

When sending, the behaviour is similar. The packet to be sent is read from a small buffer. The substitutions needed for the SLIP protocol are made and the resulting stream of data is written to the serial port byte by byte.

2.2.3 IP

2.2.3.1 Receiving

When the IP handler receives a packet from the SLIP handler, it does some sanity checks first:

- Checksum check: check the integrity of the received packet.
- Version check: check that the incoming packet is using IPv4.
- Length check: make sure that the packet and the header are at least as long as the minimal header length.
- Fragmentation check: check for fragmentations (fragmentation flag sequence ID set)
- Destination check: check if the device is really the destination of the packet (local IP address matches the destination IP address)

If any of these tests fail, the packet is dropped.
If the packet passes the tests, the byte order of the source IP address is changed from network byte order to little-endian and is delivered together with the payload to the right protocol. Currently, there are the ICMP and UDP protocols supported. Again, if the packet uses another protocol, the message is dropped.
Because the os is not meant to be used as a routing device, the hop count is not reduced or checked on incoming packets.

2.2.3.2 Sending

When the IP handler receives a packet, a standard set of headers are set and the checksum is computed. After the IP packet is arranged, it is handed to the SLIP handler. At the moment, there is no possibility for a domain sending a packet to change the headers of the IP packet (besides the destination address of course). We apply a standard set of headers to all outgoing packets.

2.2.4 ICMP

The handling of ICMP is simplistic. Currently, the only supported function is the echo reply protocol, which listens to an incoming echo request and replies with a echo reply message. Other incoming ICMP based protocols are dropped.
The ICMP handler offers a function which allows any user level domain to send arbitrary ICMP packets. However, there is currently no application making use of this functionality.

2.2.5 UDP

2.2.5.1 Open new port

All UDP ports are implemented by user level domains. A domain can open a new port by sending the appropriate message using the INMP. The domain has to implement the handler for the incoming messages and can then send a request to open a new port to the UDP handler. The handler will check if the port exists and if not, add it to a singly linked list and direct incoming datagrams on that port to the specified domain (again using the internal network message protocol). An open port can be closed again. Currently, there is no check on who is trying to close a port. An obvious improvement would be, to only allow the domain that opened the port and some monitor domain to close the ports. Also, there is currently no check, if the receiving handler of the port is still running or has crashed or has exited without closing the port.

2.2.5.2 Receiving

When the UDP handler receives a packet, it checks the destination port and forwards the packet to the right user level domain using the INMP. If the port number is not known, the packet is dropped and a message is printed for debug and demonstration purposes.

2.2.5.3 Sending

Using the internal network messaging protocol, a user level domain can send a packet to the UDP handler which writes the header of the datagram and forwards it to the IP handler. No checksum is created, because according to the RFC on UDP, the UDP checksum is optional.[1]

2.2.6 User Space Applications

Two user space applications were implemented. An UDP echo server was implemented, that automatically connects to the network process on the same core using the nameserver. All this process does, is to fetch the user input and return it to the user.
The UDP terminal server is more interesting. This application is a minimal shell, that allows the user to spawn processes and returns the output of that process to the user. It is currently not allowed to start the TurtleBack shell, as the input integration is not done. The way this works, is that the network terminal app passes a special flag to a spawning process, that signals that it should report to the network terminal. The new process will then use the nameserver to redirect its output to the network terminal. This system could be improved by integrating it to the TurtleBack shell. From a security point of view, a login mechanism should be added. By sending "exit", the UDP terminal server can be closed again.

2.2.7 Performance

2.2.7.1 Internal message handling

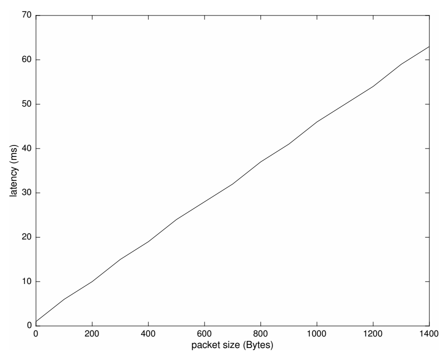
Figure 2.2 shows a measurement of the latency of the internal packet handling. For the test, the hardware counter was reset once a packet was completed. The clock cycle measurement was taken after the last byte of the message was written to the fifo of the UART device. To get a useful number out of the number of clock cycles, it was divided by the clock frequency of the processor to get the amount of ms passed. The measurement was done once using ICMP packets (figure 2.2a) and once using UDP packets (figure 2.2b). The values from this measurement steadied on a certain value after one or two measurements. We think, this might be because of allocation of additional resources (pagefault handling and transfer of ram capability), that have to be done when a larger payload is incoming for the first time.
We note, that for both protocols, the computation time is linearly increasing with the payload size. For ICMP packets, the only two operations that depend on the byte size is a memcopy operation and the writing to the fifo of the UART. As we expect the memcopy to operate in the sub microsecond regime for the tested memory sizes and we therefore conclude, that most of the time is used to write the packet to the UART fifo.

We also see, that UDP performs worse than ICMP. This is because there are two extra RPC calls (which are also payload dependant) and two context switches more involved in handling these packets.
Using the nmap tool, the latency of the response of the echo server was measured. The test was repeated 1000 times. The following values were measured:
maximum latency: 230 ms
minimum latency: 47 ms
mean latency: 68.5 ms
the measured values had a standard deviation of 21.7 ms. The packet size was 42B. This, together with the latency of the handling of a UDP packet internally, tells us that most of the time used for the transfer is actually due to the serial communication.

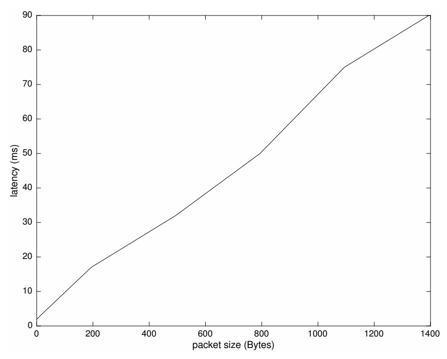
2.2.7.2 Ping

Two tests were performed to measure the performance of the implemented ping-reply mechanism. Both tests consisted of sending pings with 31 different payload sizes. The tests were run 100 times for each size. The Linux ping utility was used to perform both tests.
The first test checked the reliability of the system. For a single payload size, we send 100 packets and note, how many replies we receive. The amount of received ping replies is shown in figure 2.3a. We note, that for sizes that up to about 1.5kB, at least 98% of the ping packets generate an answer from the system. Note that on sizes larger than 1kB, some of the icmp echo-reply answers contain a faulty payload. This means, that the message sent and received do not have the same payload. However, less than 3% of the packets are affected. We see, that there were no replies of the system for payloads larger than 1.5kB. The reason is, that the packets are dropped by the tunsip utility on the ubuntu machine running the test. The error code of -7 indicates an ICMP checksum error. The error happens on the host side, before the packets reach the device under test.
In the second test, the actual RTT time for the differently sized ping requests were measured. Sizes larger than 1.5kB were excluded from the test, because of the 0 yield. The mean, as well as the lowest and the highest values for the RTTs can be seen in figure 2.3b. Different things can be noted from the data collected:

- The mean RTT is linearly increasing from 0 up to about 800B of payload. The serial connection runs on 115200 Baud. When neglecting the overhead of the slip protocol, this means that we should have a bitrate of about 14kBps. Together with the measurements from 2.2.7.1, we conclude that most of the time that is needed to handle a packet, is needed by the serial connection between the two devices and the interrupt based handling of the incoming and outgoing traffic.

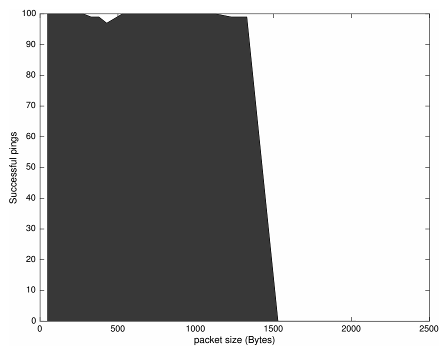


(a) internal RTT measurement for ICMP packet handling

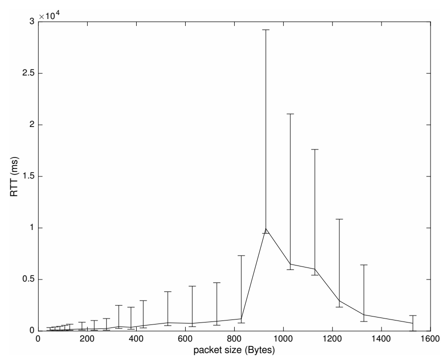


(b) internal RTT measurement for UDP packet handling

Figure 2.2: Latency between completion of packet from serial and handing over of packet to serial



(a) Successful pings (out of 100) for different ping payloads



(b) Round-trip Time (RTT) for different ping payloads

Figure 2.3: Ping tests performed using 31 different sized ping request packets.

- At around 1kB of payload, the system performed the worst. After a peak, the performance gets better again. The cause of this behaviour could not be found.
- We note, that the worst case RTT increases for larger payloads. Again, we are not sure what is triggering that behaviour.

2.3 Nameserver (Christian Fania)

The nameserver interface is organised in a fairly straightforward manner. To register, it expects you to fill out an nameserver_info struct and give it to register_service. The nameserver_info struct requires you to give it a unique name which does not contain the letter ','. Further you also need to provide a type - which can be any user chosen string that is not ','. (in both cases whitespace in the strings is fine) Which means querying on types is a good start for getting services. Further, you need to give it the current coreid, this is of little use at the moment as we will see later, but exists for a possible extension of the nameserver such that existing services do not need to be rewritten nor any adjustments to the nameserver interface is needed later. Furthermore you need to provide it with an capability which is the local cap of an open channel that accepts new connections. And then, finally, you can provide 0 to n properties, which are of the form "key:value" as they present a full on key value store. (it also means that neither keys are not allowed to contain ':', further, neither keys nor values may contain " or " but whitespace is fine). All this will then get registered with the nameserver on the register_service call - provided the name has not been used before.

The next important concept is the nameserver_query, this implements a reasonably powerful query language (though not as powerful as desired - see attached listing for originally intended query language compared to cut down version). One can query on name identity, type identity and a set of properties existing with certain keys and the values comparing to the provided values in specific ways. IN other words, every nameserver_query can have N property query objects, which state which key they want to look for (not finding it counts as nonfullfillment. Repeated mentions of the same key are legal) and then can also specify a string to compare the stored value to - either by string equality, the stored value starting with the provided, ending with it or containing it. Since this can be decided for every single query object and we can have an essentially unrestricted amount of those we can write fairly complicated queries on those properties.

Sadly, I ran out of time to properly integrate name and type into this system, in particular since other parts of the system already went gold and changing this would have constituted a breaking change for them and required a full retesting. However, they definitely should be integrated such that an uniform querying is possible (likely by having them be real or pseudo properties of the kind (name,<provided name>) and (type,<provided type>).

Once we have a nameserver_query we can then send it to lookup to get the first fit or enumerate to get the list of names of all fits. As a sidenote, by simply asking to query for props and leaving the list of query prop empty, every registered service fullfills all of those trivial and enumerate returns the full list of registered services. Since lookup returns a nameserver_info struct with all the data the original process that registered it provided, we can then do arbitrary processing on it afterwards if we feel like the provided query system is not powerful enough - though we can not write back the change, as will be explained later - thus protecting the system from some attempts at fraudulent changes. Further, the names provided by enumerate, since they are unique, are clear identifiers of the registered services and we can just send out lookup calls for all of them (or some) if we desire their actual nameserver_info. Once we have received a nameserver_info struct from the nameserver that we are satisfied with, we can then take the cap we were given there and establish a new connection with the process providing it. As an interesting side note, we do not know the true identity of the process nor do we care, as our current setup of the rpc_system is that we just establish a completely fresh channel with 2 freshly made caps on both sides (1 per side) to talk with each other.

As our memory manager, process server and serial console were never moved out of init, we just have init register all those services and the clients who get the caps to connect to those services never know on what process it actually is running, so moving those out would be an entirely transparent change from anything other than the initial set's view. Sadly, the time to move them out just wasn't there. I'll elaborate later on how it could be done.

As another sidenote, since we get all the info the original process had given to register, we could now use that cap to register a new service which leads to its endpoint and may be used to lead services astray. This, however, creates no additional security risks since:

1. if having another process send the wrong kind of things to the service we got the cap from endangers the service, we could just have send those messages directly.
2. any system can register any service currently anyway and even if it holds the end of the line for that, that can break things.
3. Just not initializing the channel the cap refers to (or sending an unrelated cap) could lead to an unsuspecting other service crashing the kernel by trying to send on that.
4. we don't even have to jump through any of those loops, just create a particularly constructed bad channel and try to send on it and thus crash the kernel on our own, without having to involve the nameserver.

What we do have, in regards to protection, however, is that a process can only deregister service that it itself has registered, this works by giving the service name to the deregister call of the nameserver. As a sidenote, the ability to register and deregister together means we do not need a call to be allowed to change properties, this can simply be done by either still having the original nameserver_info, or looking it up in the nameserver, modifying it as desired, deregistering the original and registering the changed under the same name. Since all established channels are directly with our process and not over the nameserver, this does not impact established services. Further, since we can just send the same cap we already registered there, to anyone not querying in precisely the small window it was deregistered it looks like a modify call occured.

In the future it might be worth considering to make register_service be allowed to overwrite if the original service with that name came from the same process. However this could also lead to accidental overwriting so it would have to be weighed carefully, however it seems like a worthwhile feature to me if we had the time to test (as it technically constitutes a breaking change in the general system). There we also provide a nice convenience function with remove_self, which deregisters all services registered by this process.

As for how the checking for identity works, it's actually pretty simple: The first call to the nameserver first gets the nameserver capability from init, then does a handshake with the nameserver where a new exclusive channel gets created between only this process and the nameserver. Thus the nameserver simply identifies the process by anything that comes over that channel. This has the nice side effect that simply by handing the cap to send there away it should - and this is untested as we had no use for it and it is slightly hacky - allow a process to give other processes the right to register and deregister processes in its name.

We do not have a hierarchical naming structure, as having a rich property system not only is more useful than that - ad hoc category creation and being in arbitrary subsets of them, especially once we consider the different ways we can query on - we can, if desired, also simulate a hierarchical naming structure with props, simply by adding a lvl0 prop, a lvl1 prop, etc etc which can then be understood to represent such a hierarchical tree. Sadly, automatic dead service removal does not quite work yet - services need to manually request their removal. While an attempt was made, the idea of looking for a lmp_chan changing to disconnect state as a witness of the other side being dead wasn't fruitful and then once more time constraints prevented further inquiry and approaching the topic. It would be relatively simple to have the process server notify us or something along those lines, though, as that one can successfully track processes not running anymore.

As for bootstrapping and the initial set - since we have the memory manager and the process manager still integrated into init, these just boot first and the nameserver gets its caps from there. Now, we could - with some work - make it so that init boots, then the memserver which then stays in a spinloop with init waiting for the nameserver to appear, then the nameserver could appear, register its cap with init, then wait for the memserver to register with it, immediately do a handshake with it once it does, then the memory server could keep requesting on the nameserver for the proc manager, which init now boots up, that thing gets the nameserver cap, gets the memserver cap from the name server (or waits until both of those are available if it boots too quickly), then could register itself with the nameserver and then the nameserver and the memserver could register themselves with the proc manager and all three come fully online at essentially the same time, with only the nameserver cap having ever had to go to init. This would be one way - the absolutely minimum "going over init" way of solving it. Another simpler way would be to have an initial set of processes which just work like this set would have worked without the existance of a nameserver (this likely would at least have to be init+memserver+nameserver, if not also the procmanager, depending on implementation details) and then the namserver just gets added either after memserver boot or after proc manager boot.

Back to our actual system - once the nameserver is up, all other processes do get their memserver, proc manager and serial console caps from the nameserver. Further, the network system uses the nameserver to allow its various parts to establish intercommunication. However, one core limitation of the nameserver as it is currently - primarily due to time reasons and us never having constructed an unified RPC and URPC system - core specific. That is, it can only provide connections to other services running on the same core. As our system is mostly a bipartited system with most functionality replicated across cores to some degree or another, this is actually somewhat helpful, as it means every process will be guaranteed to get the correct memory manager.

However, if time had permitted it, one simple way to improve on this would have been to extend the nameserver and the proxy server in such a way that the nameserver replicates service entries across cores unless they are marked as core exclusive by using an extended proxy server which is extended in such a way as to allow us to fake a lmp_chan across URPC, by opening a lmp_chan on the other side, being able to deal with chan spawning via handshakes and just tunneling all data over URPC. Then the nameserver on the core the process was not originally from, instead of having the actual process's chan capability in its entry, has one generated by the proxy server which is set up such that it will direct any calls to the original process. This is a bit tricky to make work with proper handshake propagation (as it spawns new channels), but should still be doable as the proxy server initiates it on one end and knows where it came from on the other end. However, sadly this did not make it past the planning stage due to time.

```haskell
--in theory this could be further improved by moving QName and
    QType in as pseudo props or actual props as described in the
    text
data Query = QName (Ops1 OpsString) | QType (Ops1 OpsString)
     | QComplex Query2
data MetaOps a = Negated MetaOps | Any [MetaOps] | All [MetaOps]
     | MetaOpsObj a
data OpsString = Is String | BeginsWith String | EndsWith String
     | Contains String
data Query2 = Q2 (Ops1 (OpsString,Ops1 OpsString)])

data Namserver_info = NSI String String [(String,String)]

compare (NSI name _    _     ) (QName x) = metaOpsCmp
    opsStringCmp name x
compare (NSI _    type _     ) (QType x) = metaOpsCmp
    opsStringCmp type x
compare (NSI _    _    props) (QComplex y) = metaOpsCmp c3 props
    y

foldr (&&) True $ map (c3 props) y


opsStringCmp x (Is y) = x == y
opsStringCmp x (BeginsWith y) = (take (length y) x) == y
opsStringCmp x (EndsWith y) = x == reverse (take (length x) (
    reverse y))
opsStringCmp x (Contains y) = IsInfixOf y x

metaOpsCmp f x (Any ys) = foldr (||) False (map (metaOpsCmp f
    props) ys)
metaOpsCmp f x (All ys) = foldr (&&) True (map (metaOpsCmp f
    props) ys)
metaOpsCmp f x (Negated y) = not (metaOpsCmp f x y)
metaOpsCmp f x (MetaOpsObj y) = f x y

c3 [] _ = True
c3 (name,attr):xs (name2,query)
  | name == name2 = metaOpsCmp opsStringCmp attr query && c3 xs
    (name2,query)
  | otherwise = c3 xs (name2,query)
```

Listing 2.1: Originally intended Query System (haskell pseudocode)

Bibliography

[1] J. Postel. User datagram protocol. STD 6, RFC Editor, August 1980. http://www.rfc-editor.org/rfc/rfc768.txt.