Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Project Report

Advanced Operating Systems GroupF

Autumn Term 2017

Supervised by:
Timothy Roscoe
David Cock
Simon Gerber

Authors:
Victor Chibotaru
Alexander Hedges
Christopher Signer
Zhifei Yang

Contents

Chapter 1

Group work: Building the core of the OS

1.1 Memory management and capabilities

Memory management is mostly a tradeoff between wasted space and fragmentation. Fragmentation occurs when the amount of requested memory is available but not in a contiguous range. On one extreme, if you always hand out the requested memory and break it off from the next available block (either using best possible fit or worst possible fit) it is possible to construct an allocation pattern for which this scheme heavily fragments the memory space and is either not able to return the size or has to do expensive compaction before. The other extreme case is a constant size allocator (let's say it always returns 4k of memory). This scheme has the advantage that there is no fragementation, but on the flipside this is very wasteful if the requesting caller only wants a few bytes for each allocation. It is also a problem if a size larger than the block size is requested. We chose an approach that lies somewhere in between the two. We always return a piece of memory that has a size which is the smallest power of two that is larger or equal to the requested size. Like this the wasted space is upper bounded by the used space.
A constraint we faced was that the call we implemented also takes an alignment parameter. If we were to honor arbitrary alignment parameters, which we tried to do first by cutting off the parts from the region at the beginning until it was aligned, this could again result in terrible fragmentation (assume the user requests a piece of memory to be aligned to a large prime number). Also you would have to over-provision the size since you also have to take alignment into account.
Our first implementation also had the flaw that it could not merge adjacent free memory regions once a key piece was returned to the memory allocator (this would have required to do some expensive walking of the circular linked list). Eventually we decided to a rewrite it to correct these issues. We used what is referred to as a buddy system. Conceptually the space is partitioned into regions that have sizes which are powers of two (except for possibly one). If no region is small enough we can recursively split a region into half until we get a region of the desired size. This makes insertion easy since we can calculate with which other memory region (the buddy) it should be merged. We stored the regions sorted by size in a circularly linked list to make retrieval of the smallest fitting node easy.
We solved the alignment problem by only guaranteeing that we will return a memeory region that is aligned to the constraint if the constraint is a power of two (like libc does too). The only problem we faced with the new approach was that this method is very, sensitive to the starting address of the initial region, if it is not a

sufficiently high power of two, every alignment is squewed by the base address. We solved this problem by first splitting off a small chunk of memory (for which we could not guarantee any alignment properties) so that the remaining part is aligned to a larger power of two.
This approach is a reasonable tradeoff between possible fragmentation and returning too much storage.
Using capabilities to represent the memory regions added some intricacies, also since it took us some time to wrap our head around the capability system. First of all you cannot split and merge two adjacent capabilities. What you have to do is to retype a child capability with a certain size and offset into the parent and return that. Then you have to do some bookkeeping about the actual vs nominal region the capability refers to and make sure you don't hand out the capabilities which refer to more than they say the hold.

1.2 Message passing

Message passing is the essential communication between processes. This task had a specific focus on lightweight message passing (lmp) by implementing a remote procedure call (rpc) interface that creates comfortable usability for the communication between a child processes and the init domain. User-level message passing 1.6 will go on to allow more flexible communication specifically also across cores and the nameserver 2.4 will allow for easier binding.

1.2.1 Initialization

The aos_rpc interface is supposed to fit onto the Barrelfish Kernel's LMP channel functionality which can send a capability and 9 words per lmp message. A first channel is built using lmp_chan_accept to connect from the spawned process to the init process by sending a message to the init endpoint which is globally available. The message contains the capability endpoint of the child process that will then allow the init domain to send messages to the child process and in particular let the init domain respond to the child process that it received the message and the connection works. Both processes can store the state now to connect to send without further setup again and to also setup a receive handler. The receive handler is registered on the lmp channel together with a waitset. The waitset will allow for the receiving process to check when scheduled upon which the receive handler can be executed to handle the rpc message.

1.2.2 Message protocol

To identify messages, we introduce an enumerable called enum_rpc_msgtype which is sent with every message to identify the purpose and format of the message. Messages can be send directly and the process can be setup to wait for an answer synchronously, but for most purposes we use internally methods like send_and_receive to send a message and register a specific receive handler that will handle the answer once arrived in an asynchronous way, thus allowing the process to handle other messages meanwhile and possibly avoiding discarding messages. Acknowledging messages could have the advantage of allowing us to send especially low priority messages that have to be split up in multiple lmp messages not all at once but always after receiving an answer from the recipient, to thus reduce the load on the recipient in case of congestion at handling messages, but more detailed analysis and actual implementation of prioritizing messages were out of scope.

1.2.3 Interface, message marshaling and receive handling

The given interface consists of standard rpc calls to send common data types to a specific channel, i.e. aos_rpc_send_number, aos_rpc_send_string and similar. Especially strings and similar buffers or other data of variable size are more challenging as they have to be possibly be split on a large number of lmp messages. We thus created rpc calls to fit our needs and fit data in efficient ways into lmp messages. In hindsight this marshaling of arguments could be much more generic though, to reduce the code size immensely and make it much easier to use. How the data is split into messages of 9 words (and possibly a capability) each could be left largely they same. We attach to each message the rpc message type as a first argument, then more fixed size data, before a length is given of how much variable size data is attached and due to follow in the following messages. The message type identifies what format the data has and thus allows to reconstruct the sent data into the original data formats. This is done with one general receive handler that calls the right secondary receive handlers which process the rpc call according to the message type. The answer arguments are constructed at that point, before the lmp channel register send is dispatched with an appropriate send_handler, analog to the initially sent rpc message. Messages with capabilities have to be treated specially as the recipient requires a free slot to receive the capability but are otherwise analog.

1.3 Virtual memory, lazy allocation and page faults handling

1.3.1 Virtual memory management

The main building block for keeping track of allocated virtual space chunks (regions) is a data structure named *paging_region*, which contains info about the offset of the start of the chunk and its size.

These regions are organized in two lists: one list is for the free ones, the other for the taken ones. The "free" list is ordered by the regions' offsets to facilitate defragmentation, the "taken" list is unordered. Pointers to the heads of both lists are stored in *paging_state* (the main data structure responsible for paging).

Whenever some amount **X** of virtual memory is requested (via a call to *paging_alloc()* function) the following steps are taken:

1. A region of size at least $\mathbf{Y} \geq \mathbf{X}$ is searched in the free regions lists.
2. If such a region is found, then it is split into two subregions:
   (a) First subregion of size **X** is added to the list of taken regions. This subregion gets returned to the caller.
   (b) Second subregion of size **Y-X** is added to the list of free regions (actually the original region's size and offset are modified)
3. If no such region if found, then an error is returned.

An allocated region can be unmapped via a call to *paging_unmap()* function. The algorithm is pretty straightforward:

1. Find the region in "taken" list using the offset provided.
2. If the region is found, then:

   (a) Move the region from the "taken" list to the "free" list.
   (b) Perform defragmentation by joining the adjacent free regions (*free_list_defragment()* function).

3. If the region is not found, then return an error.

The choice for the data structure in the case of free regions tracking seems pretty natural - lists allow easy traversal, region splitting and, the most important, defragmentation in linear time by gluing the adjacent regions. In the case of taken regions a hash table or a RB tree would allow faster search, but we have chosen lists because of implementation issues.

1.3.2 Lazy allocation and page faults handling

One of the project requirements was to implement lazy memory allocation. The main idea of it is to only allocate virtual (and not physical) memory whenever a memory allocation request is made. Then, if the memory is actually accessed, a page fault occurs (because the virtual page is not mapped to any physical page) and gets processed by our page fault handler. The handler (*exception_handler*) would then allocate some physical memory and map it to the virtual page, which triggered the page fault.

One of the main things needed to make page fault handlers work is allocating an exception stack. We do use a static buffer for the main thread of each domain because the exception stack is set up before the virtual space tracking data structures ("free" and "taken" lists). For the others threads the stacks' addresses are allocated using the virtual memory management functions and are immediately mapped to physical frames.

1.4 Threads

To make multithreading work properly we had to make sure that all the shared data structures are thread safe:

1. *struct paging_state* - this structure contains two mutexes, which guard access to two different entities: one is the "regions data" (the "free" and "taken" lists; this mutex is deceivingly called *paging_data_mutex*) and the other is "page mappings data". In the beginning we tried to make two separate mutexes for each of the lists to make things more modular and faster. This implementation soon proved to be way too complex and hence we decided to lock and unlock the *paging_data_mutex* in the beginning and the end of all publicly accessible functions. This approach is much simpler and provides the needed level of synchronization.

2. *struct mm* - the memory manager. For it we did the same trick as with *struct paging_state*.

3. *struct slab_allocator* - the slab allocator was not thread safe out of the box, so we had to add a mutex to it.

4. *struct slot_allocator* - the slot allocator is a fairly complex object, which, thankfully, had the needed synchronization mechanisms built in.

   For all the data structures described above implementing thread safety implied choosing between making the synchronization rules more complex and granular, which theoretically should result in better speed or having them

simple, few and reliable, which results in slower, but more secure and safe access. We sticked with the second option. The tests we used for threads are located in *usr/init/tests.h*.

1.5 Multicore

The mulicore milestone was conceptually not that difficult, but it involved a lot of steps that had to be right otherwise it would not boot and it was quite impractical to debug.
Basic inter-core communication was sort of required so we already mapped a shared frame into both processes and done a very basic communication sceme (looking at status flags), but then it was announced that this would only have to work by the time we do the ump.
To also give the other core memory it can access we did a three way split of the memory (one split was already described in the memory management description) and passed a suitable (high power of two aligned memory) via the shared frame to the second core. We used the first word of urpc frame as a status indicator. After boot_core(1), core 1 keeps polling on the status, and core 0 writes its bi-¿mem_region structs to the very beginning position of the shared frame and updates the status, so that core 1 can start to read the information to get memory layouts. After finishing reading, it updates the status (so that the urpc frame can be used to pass other messages, in our implementation, the remote spawning message), and adds the second half of every memory region to its memory manager.
Later with full ump working we just used that to transfer the the address range to the second core where it could forge the capability belonging to it.

1.6 User-level message passing

This milestone dealt with inter-core communication. The most important use case of inter-core communication is as an extension of lmp but where the message has to arrive on another core.
This can be made very fast since both ends can be running at the same time without having to be interrupted. The idea behind it is to spin on a shared cache part and using the cache-lines as a circular buffer.
Since we spin on two cache lines at a time (64 bytes, that is 16 words) we have more space than in a single lmp message. To simplify matters we are encapsulating the lmp messages in a ump message and don't use the remaining 5 words. So for a program a on core 1 to communicate to program b on core 0 the message is sent to init on core 1 which sends it to core 0 via ump. There the message is unpacked to and sent to process b via the lmp handle that init has on core 0. Init acts as a central communication point that forwards traffic between processes. In lmp the destination is encoded in the channel. Since both cores only have one incoming channel the first word in the ump message is reserved for the destination process id (init has process id 0).
One thing to look out for was inserting barriers at the right places so that the reading and writing happened after the data the message contained became valid.

Chapter 2

Individual projects: Building on the core

2.1 Shell

This part was about implementing a process that interacts with the user and provides a way of testing the system. It was implemented by Alexander Hedges.

2.1.1 Character input

This part was about getting a character from the UART device and sending it to the requesting user process.

Roughly this involves setting the interrupt handler to call a certain function everytime the keyboard is pressed. This is done in the terminal files in the init process. Characters are buffered in a circular buffer of size 1024 until a newline (actually carriage return) is recognized and then sent back if a receiver has registered for the input. If no receiver is registered it just keeps buffering characters. Right now there are no protections against overflowing the buffer but since the buffer is circular and a null character is always inserted after the current char the implementation should be fine (at least from a security perspective, the better thing might be to disable moving with input past the current_read_index).

String input and ouput uses serial_getchar and serial_putchar which is slower than using the respective methods for sending strings, but having IO work on characters is trivial while strings can be more tricky to get right and might have edge cases.

For this to work like in scanf it is only allowed to send back characters when the user typed a carriage return (enter). So instead of doing some busy waiting on the input, the init process saves a callback from the last process that requested the characters and uses that to send back characters to the place where they are expected. Note that reading characters also works across cores since it can simply provides a different callback to init.

In order to be useful the terminal service of init also has to do some limited processing of characters. This includes changing carriage returns to linefeeds. One other thing which is done here is reducing the amount of characters in the buffer everytime we get a DEL control code. In order to remove the already printed output it does some backspace and clearing magic.

2.1.2 User interaction

This part was about writing the userspace program which acts as a shell. Fortunately, since we implemented the first part, the shell can be written like a regular c program with the only difference that is makes use of the rpc library from aos. Here I also noticed that the first version of our character input was working in a way that `scanf` didn't do the same thing as its counterpart on the host OS. This also enabled faster prototyping.
The shell doesn't include any fancy functionality (apart from character deletion which I implemented 20 minutes before the demo session).
The reading a line happens via fgets where we can make sure our 1024 static buffer is not overrun. To facilitate parsing there are the `parseline` and `token` structs. Some useful helper functions are implmented to tokenize the input and to output errors. Tokenization is not perfect since it handles strings, but not escaped characters so it is impossiblle to print a `"`. Escaped characters are a bit more tricky since they require modifying the whole line (replacing the escaped char and then shifting all the content to the left by one character). If I had more time I would also have implemented a convenience function to match a certain input string (right now this is quite verbose).
The (useful) supported builtins are `echo`, `daemon` and `run`. Echo closely matches the behavior of the real bash echo which prints its arguments separated by a space. `daemon` and `run` both are responsible for spawning processes but while daemon spawns the process in the background and keeps going, run is supposed to wait for completion. Run is needed to spawn processes that require user input from stdin. As described, user input works on a last-come first serve basis so it is not entirely clear what would happen when two programs are simultaneausly competing for user input. The most likely scenario is that since the shell queries the input continuously it would starve any other kind of program from stdin access. The first argument of the spawn commands is the core the process should be spawned on.
Right now `run` does the wrong thing of putting the program into an endless loop in order for the child to gain "exclusive" access to the standard input. The right thing would be to wait somehow for program completion but since to my understanding does not happen in our Barrelfish version it effectively does the same thing (probably then it would be better to at least yield in the while loop).
Note that while both daemon and run could easily be modified to pass in multiple arguments to the shell, the spawn process rpc does not support it. Implementing this would have been my number one priority given more time. Right now we have one-to-one, one-to-many (one-to-many in terms of sending one lmp message and getting multiple messages back as an answer) and asynchronous messaging (`serial_getchar`) working also across cores, so many-to-one messaging would be the missing piece to a complete lmp/ump implementation.
The "desperate" help program also advertises `ls` and `ps`. Ls does not work because the filesystem does not implement everything yet that is necessary for `ls`. The process list command, which was taken from an earlier milestone test used to work but before the demo it seems to have been broken by one of the changes. But I guess this is what you get when you have three people concurrently modifying the source code until right before the demo.

2.2 Filesystem

This part was about adding file system support for a FAT32 formatted SD card and was implemented by Christopher Signer.

2.2.1 Development process and system design overview

The initial goal was to allow at least read operations on the SD card by adding a filesytem server (referenced as fileserver in short in the code at some places) and connecting it into the system such that the libc library calls use it and it works together with the given mmchs block driver. The fileserver would be an own process, running ideally on one core but handling rpc calls from processes on both cores through the urpc interface. This would also remove some problems of concurrency as the fileserver would receive one rpc call at a time and thus handle them sequentially. At the same time it was the initial idea to run the mmchs commands directly on the fileserver process as well, reducing the overhead of more rpc calls between the fileserver and mmchs, especially since these rpc calls would involve transmitting large amounts of data, but as a downside preventing the fileserver from working on processes while doing read or write operations on the SD card.
The implementation of the /lib/fs/fat32 library (filesystem functions that the fileserver would use) and the /lib/fs/fopen_fat32 (libc hooks) turned out to be quite a bit of work, although the interface and structure could be kept similar to ramfs and fopen. The data structures handling and actual fat32 implementation were the significant change there. The rpc calls to communicate with the fileserver then turned out to require a large effort due to the more complex data structures that should be sent across rpc which our design didn't support yet well. At this point refactoring would have been more even more work than ealier on as it was already more integrated in the other services. Thus sticking to the rather messy rpc design together with unsuccessfully trying to include the mmchs functionality in the fat32 library (fiddling with hake) cost me time which I then didn't have anymore when I decided to run the mmchs block driver as an own process with it's own rpc calls for reading due to a lack of a better alternative. And after all, this would have been the essential part to get working to test whether the filesystem library actually works. Thus in conclusion there are lots of fragments, especially in the fat32 library which should almost do what they should (as it's not tested yet), but as a whole the fat32 filesystem cannot be run. The following will thus partly describe how it could or should be, but not only working implementation details.

2.2.2 Block driver server

The mmchs block driver when started as process first calls given initialization methods and then init_service() of the service_aos.c where the process can register with the nameserver before registering the receive handle to the lmp channel to the fat32 fileserver (usr/fileserver...). For read only there's one essential rpc call required that takes the block number to read, calls the mmchs_read_block method to read 512 bytes into the buffer and returns the result back to the caller (fileserver). It might obviously make sense to restrict who may call the rpc call for the write method to prevent the filesytem from getting corrupted.

2.2.3 Fileserver

The fileserver consists of another process that is registered with the nameserver and handles rpc calls for the filesystem operations. The filesytem operations could be called directly or via libc functions (see 2.2.4).

fat32 library initialization and data structures

The idea was to have a static singleton struct fat32_mount that is instantiated once during the mount and then available through rpc calls by all processes as there's only one SD card and any mount location should be available to any process. To be more flexible it might be a better service to be able to mount the same SD card to multiple locations. The mount instance should regardless still be the same except for the path, to use the available hardware resources better and provide better performance (as we especially share more cached data). This would also require better integration of fat32 with ramfs as a mount path then could also be in any folder of ramfs.
The mount call reads sector 0 from the filesytem by using the mmchs read rpc call, reads the various parameters of the fat32 into the mount singleton and of course checks whether it actually is a valid fat32 filesytem. It specifically also creates the handle to the root directory and loads the cluster thereof. It also allocates the fat allocation table which is filled lazily with cache values during usage.
Analog to the ramfs_handle, the fat32_handle stores also the name, file position, a pointer to a fat32_dirent and whether the handle points to a directory or a file. Additionally it also contains a pointer to the cluster data if the handle is a file and the data is in memory. For directories that field contains a pointer to the current fat32_dirent directory entry, which contains the directory attributes and name. It also contains the current offset of read bytes in the current cluster for files, respectively the offset to the dirent in the current_clust if we were to open the directory of the handle, while first_clust contains the cluster number at which file / directory of the handle's data starts at.

fat32 library features

To open a directory we start off from the root directory and match the path to the directory we wish to open by looking at the dirent of the root directory which contains the cluster in which we can look at the directory entries in root and their names. In case of a match of the prefix to the next forward slash of the path, we look at that directory's entries, combing through the cluster that this new entry specified, and so on. These clusters are cached in the fat allocation table and read only from the SD card when needed to prevent a long setup time in the beginning and to improve the performance at the cost of some memory to improve access times. Further we provide methods to close a directory handle and read the name of the next directory entry (if available). We also offer fat32_stat which takes a handle and returns a fs_fileinfo struct that contains the size and whether the handle points to a directory or a file. The closedir method frees the memory held by the directory handle.
For files we offer also opening, closing handles and getting the file information, which work in similar fashion. Additionally we can read files and seek, thus change the file position offset to either an absolute value from start, a relative value from the current position or an absolute value from the end of the file. Read quite straight forward takes a file handle, a number of bytes to be read, a buffer of that size and a pointer to a size which is returned. Read first determines whether the full amount of bytes asked for can be read or whether the file ends earlier. It then loads the determined number of bytes from the file into the buffer and moves the file position appropriately. It might have to load more clusters than the current one if the file is spread across multiple clusters and returns the number of bytes read. Seek determines what the new file position offset should be and changes the file handle accordingly to point to the correct clusters. It returns the new file position offset.

2.2.4 Connecting the fileserver rpc calls to libc

Analog to the given fopen.c, fopen_fat32.c can connect the libc functions to the appropriate rpc calls to handle data. filesytem_init in fs.c is changed to call fs_init_libc which uses an fd table to call the right handler for the libc functions, call the rpc file function and return the answer. Besides not linking the unsupported functions this replaces the linking of the ramfs methods. A nicer solution might be to merge the two and determine for each call when the appropriate handler of the libc method is called whether it's a ramfs or a fat32 call and then execute the right method.

2.3 Networking

This part is about developing a basic network stack for our operating system. It was implemented by Zhifei Yang.
From a TCP/IP protocol stack perspective, what we have built are the layers above the physical layer. The Physical Layer is already provided by the user level UART driver, which uses UART4 of the OMAP44xx SoC, and we only need to brining it up. Based on this we build the Link Layer, for which the Serial Line Internet Protocol (SLIP) is implemented. The SLIP protocol will decode SLIP-encoded datagrams and forward it to the next layer: the Internet Layer, which is handled by the Internet Protocol (IP). IP protocol will identify the datagram by the Type of Service (TOS) field in the header, then forward it to the matching Transport Layer protocol, or process the packet by itself (e.g. for ICMP packets, as elaborated in **??**). For Transport Layer protocols, due to time constraints, we only implemented the User Datagram Protocol (UDP). UDP will identify the destination port and look up if there is a network client listening on the port; if so, it will forward the port to the client. The client is also able to send out a message via UDP.
Figure 2.1 illustrates the hierarchy of layers and corresponding protocols we implemented. In our case where SLIP is used as the Link Layer protocol, there are no frame headers.

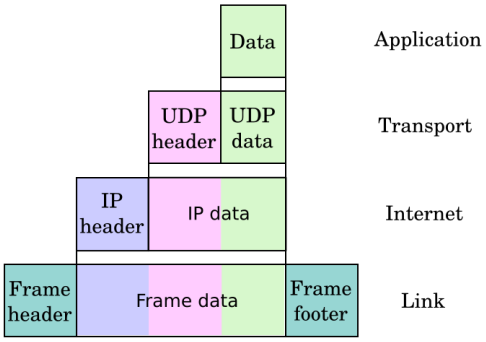

Figure 2.1: Protocol Stack

In our system, we build networking as a service, which means that the networking process is registered in the nameserver as a service provider. The networking process starts with the following steps:

1. Bring up the user-space serial driver for UART 4, which allows us to communicate with the host machine on the serial cable.
2. Set up a RPC server to serve port-binding and packet-sending requests.
3. Register itself as a service to the nameserver, so that other process can get the endpoint capability to networking process.
4. Enter the message handler loop to wait for events, such as RPC calls.

2.3.1 Bringing up a second UART driver

To bring up the second UART driver, first, we need to get a device frame capability to UART4. For this, we implemented a new RPC call: `aos_rpc_get_device_cap`. The networking process gets the device frame capability from `init` by this call.
Then, the frame is mapped into our own address space and the `serial_init` function of user space serial driver is called to initialize the driver and register the interrupt handler.

2.3.2 Implementing the protocols

We implement SLIP with a buffer length of 1006, which is the standard buffer size. Every byte we get is put into the buffer, until we receive an `SLIP_END`, after which `slip_packet_recv_handler()` is called to process the packet.
There are nothing interesting to explain for IP and ICMP protocols. For UDP protocol, the checksum is skipped by default because we run it on IPv4, which allows an empty checksum. But we have also implemented checksum support, which can be enabled by undefining the related macro in `slip.c`.

2.3.3 Providing networking service for different processes

To provide networking service for different processes, a service is registered to the nameserver. The client should be able to handle DELIVER requests from network, while the network can handle INIT and FORWARD requests from clients.

Handling RPC calls

2.3.4 Measuring the performance

Bandwidth

To test the maximum bandwidth of our networking implementation, we flood the network by sending fixed-size (84 bytes including the header) ping packets in dimidiate intervals step by step, starting from 1 second. Table 2.1 shows the experiment results.

Ping Interval (ms)	Bandwidth Equivalent (Byte/s)	Latency (ms)
1000.000	84	32.559
500.000	168	32.230
250.000	336	33.038
125.000	672	32.761
62.500	1344	33.040
31.250	2688	31.695
15.625	5376	31.872
$\leq$ 10.000	Fail	Fail

Table 2.1: Bandwidth measurement by changing ping interval

As we can see, when the ping interval is less than or equal to 10 milliseconds, the networking process fails after replying to two packets. The reason is that when the bandwidth exceeds some threshold between 5376 Bps and 8400 Bps, the serial line gets saturated, and some bytes are dropped under this high load, and the packet received on an `SLIP_END` becomes incomplete. As a result, the completeness checking assertion for IP packets fails, and the process gets stopped.

Latency

To test the latency of the network, as well as its relationship with packet sizes, we ping the system with custom-sized packets and record the average latency as well as the standard deviation. The experiment data is plotted in Figure 2.2.
As the figure shows, the network latency grows linearly with respect to packet size, which is understandable because the packet processing can be considered as a O(N)

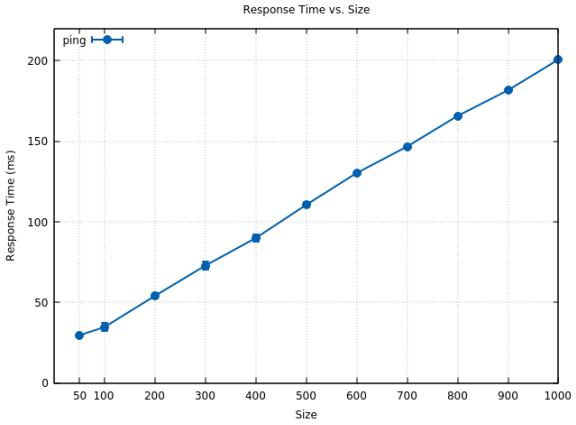

Figure 2.2: Network Latency as a function of Packet Size

algorithm. There is also a intercept on the latency axis, which refers to the base overhead for receiving and buffering the complete packet.

2.4 Nameserver

The nameserver (further called simply NS) is a domain spawned by init, which allows other domains to register their services and lookup services provided by others. The NS was implemented by Victor Chibotaru.

The NS infrastructure consists of several pieces:

1. The NS domain (*usr/nameserver/*)
2. The NS client library (*include/aos/nameserver.h* and *lib/aos/nameserver.c*)
3. The RPC functions *aos_rpc_*_nameserver_** (*include/aos/aos_rpc.h* and *lib/aos/aos_rpc.c*)
4. A few RPC handlers in init (*usr/init/lrpc.h* and *usr/init/lrpc.c*)

2.4.1 Bootstrap

The first issue one needs to solve is the bootstrapping problem - how can domains get the endpoint capability to interact with the NS? An obvious solution would be to make init place this capability in a well defined slot in the newly spawned domain's CSpace. Init does the same thing with its own capability. The cons for using this approach are the following:

1. If a domain is spawned before NS, then this domain has no way to find out about the NS's endpoint.
2. If the NS crashes and has to be restarted, then all of already spawned domains will not be able renew its endpoint.

Our solution was to make domains obtain the NS's endpoint from the init domain via RPC (see *aos_rpc_get_nameserver_ep()*). Accordingly, after spawning and initializing its listening service, the NS has to register itself with init (see *ns_init_rpc()*).

2.4.2 Channel to NS

After obtaining the NS's endpoint a domain should initialize an lmp channel to it. To do so, the client can invoke the method called *ns_init_channel()*. Under the hood this function does an AOS_RPC_ID_INIT RPC to the NS. After receiving such a message, the NS stores the channel back to the client, issues an unique client id and sends it back to the client. For all further interactions the client is supposed to provide the issued client id with all the messages it sends to the NS.

The client id is needed mainly for authentication and authorization reasons. We do realize however that the current implementation of short predictable numeric client ids is not really secure. A safer alternative would be: longer random integers together with defense against bruteforce or e.g. using capabilities as authentication tokens.

Another handy usage of client ids is storage of clients' state objects (e.g. the lmp channel, cur transmission buffer, etc.). For more details, please see *usr/nameserver/whois.(hc)*.

2.4.3 Register

For registering a service with the NS, the client can use the *ns_register()* function. To do so he has to provide an endpoint capability for the service and a service name. Since the name can be arbitrarily long the request may require sending many RPC packets. The marshaling and unmarshaling are done in *aos_rpc_nameserver_register()* and in *ns_marshal_register()*.

The data structure used for storing the services is a list (see *usr/nameserver/names.(hc)*. A faster alternative would be using a hash table or a balanced binary tree, but due to lack of time and a standard library, we decided to stick we an easier and reliable solution.

2.4.4 Deregister

Deregistration is really similar to registration, except we delete a service instead of adding it.

A feature that the current implementation is lacking is authorization. The NS does not verify that the client who is deleting the service is the same client who added it. This might lead to DOS attacks in a real OS.

2.4.5 Lookup

By using the *ns_lookup()* function, a client is able to retrieve an endpoint capability for a service whose name matches the given query. Again, as the queries can be really long, the implementation supports sending an arbitrary number of RPC packets. The NS returns the endpoint only if the query exactly matches the name of one of the services.

2.4.6 Enumeration

By using the *ns_enum()* function, a client is able to enumerate all of the services registered with NS. The list of services is sent as a string, obtained by concatenating all of the names and a specific delimiter. This might pose a problem, because the names are not verified at registration time and they might contain the delimiter, which would make unpacking the list problematic.

2.4.7 Examples

For usage examples please see the domain *calc_server*, *hello* and *ns_client*. The first two are the client and server counterparts for a calculator application, which computes and prints the values of arithmetical expressions. The third one is an utility to enumerate and print the services registered with the NS.

2.4.8 Conclusion

Our implementation of the nameserver is fairly simple and definitely lacks a number of features. Nevertheless it provides all of the basic functionality required and is pretty robust. Due to lack of time we had to choose between making the basic features work properly or add new features at cost of reliability. We have chosen the first option.