Karlsruher Institut für Technologie

A Naming Scheme for libRIPC

Bachelorarbeit
von

Andreas Waidler

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Hartmut Prautzsch
Betreuender Mitarbeiter:	Dipl.-Inform. Jens Kehne

Bearbeitungszeit: 12. Juli 2013 – 11. November 2013

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 11. November 2013

Abstract

Future cloud scenarios are expected to be realized in a fashion similar to current high-performance computers. Instead of using few powerful nodes that are connected by a simple network technology, they are expected to consist of low cost nodes connected by a network that provides features like user-level I/O or remote DMA. Processes are distributed individually among these nodes and can migrate from one node to another, for example due to hardware failure or load balancing.

Efficient inter-process communication requires using operating system or library functions that take advantage of advanced features of the network and support persistent addressing of migrated communication partners. To make full use of advanced features, current high-performance network architectures require communication to be asynchronous and message-based. However, current operating systems rely on sockets for remote inter-process communication. Sockets have synchronous and stream-based semantics, and identify processes by their physical location. Thus, communication mechanisms of current operating systems are not suited for future cloud applications.

LibRIPC [4] is an inter-process communication library designed for such cloud scenarios. LibRIPC provides an interface compatible to the underlying network hardware and improves performance of cloud applications by taking advantage of the network's advanced features. To identify processes, it uses hardware-agnostic service IDs which it resolves to network addresses by using a simple broadcast-based scheme. This approach has several disadvantages. First, service IDs are mere integers and hard for humans to remember. Secondly, broadcasting resolution requests creates communication overhead for all processes in the network. Thirdly, any response to a resolution requests has to be trusted. Lastly, there is no mechanism that a process could use to notify communication partners about the fact that it just migrated to a different host.

This thesis presents a naming scheme for libRIPC that allows processes to be identified by user-chosen service names instead of service IDs, and a resolution mechanism that reads network addresses securely from a central directory in the network. Lookup results in a capability object, which our naming scheme uses to address communication partners. By sending notifications when a service

migrates to a different host, we enable libRIPC instances of client processes to update their capabilities. By keeping capabilities updated, we ensure that processes can address services using the same token, regardless of whether the service has migrated.

We describe how we implemented a prototype using Apache ZooKeeper [3]. While evaluation of this prototype against the current resolver of libRIPC showed that ZooKeeper-based resolution of service names takes about 35 times the time it takes to resolve a service ID by using InfiniBand broadcasts, addressing processes via capabilities instead of service IDs proved to add no overhead.

Deutsche Zusammenfassung

Kommende Cloud-Netzwerke werden voraussichtlich ähnlich wie aktuelle Hochleistungsrechner aufgebaut sein. Anstatt weniger, rechenstarker Knoten werden voraussichtlich viele leistungsschwächere aber effizientere Prozessoren verwendet werden. Einfache Netzwerkhardware wird voraussichtlich durch Hochleistungs-Netzwerke ersetzt werden. Diese Hochleistungs-Netzwerke bieten Möglichkeiten wie user-level I/O oder remote DMA. Wegen möglicher Hardware-Defekte oder Load Balancing kann sich die Adresse eines Dienstes ändern.

Effiziente Interprozesskommunikation in kommenden Cloud-Netzwerken erfordert die Verwendung von Betriebssystem- oder Programmbibliotheks-Funktionen, die besondere Möglichkeiten der Netzwerkhardware nutzen und es Nutzern erlauben, Prozesse zu adressieren, unabhängig davon, wo sich Sender und Empfänger im Netzwerk befinden. Für bestmögliche Performanz auf aktuellen Hochleistungs-Netzwerken müssen Schnittstellen für die Interprozesskommunikation für den asynchronen Versand von abgeschlossenen Nachrichten ausgelegt sein. Aktuelle Betriebssysteme benutzen jedoch Sockets, welche synchrone Kommunikation über Datenströme realisiert. Außerdem identifizieren Sockets Prozesse über ihre Adresse im Netzwerk. Daher sind Kommunikationsmechanismen aktueller Betriebssysteme nicht optimal für kommende Cloud-Netzwerke geeignet.

LibRIPC [4] ist eine Kommunikationsbibliothek für kommende Cloud-Netzwerke. LibRIPC stellt eine Schnittstelle bereit, welche kompatibel zur Semantik der unterliegenden Netzwerkhardware ist. LibRIPC nutzt die Möglichkeiten der Netzwerkhardware und verbessert so die Performanz der Interprozesskommunikation. Um Prozesse zu identifizieren verwendet LibRIPC hardware-unabhängige Service-IDs. Diese Service-IDs löst LibRIPC mittels Nachrichten an alle Prozesse im Netzwerk auf. Diese Herangehensweise an die Adressierung von Prozessen hat Nachteile. Erstens, Service-IDs sind lediglich Ganzzahlen und lassen sich von Menschen daher nur schwer merken. Zweitens, Auflösung von Service-IDs mittels Broadcasts verursacht unnötigen Mehraufwand für alle Prozesse im Netzwerk. Drittens, jede Antwort auf einen Broadcast zur Namensauflösung wird von LibRIPC als authentisch angesehen. Desweiteren bietet LibRIPC keine Möglichkeit, Kommunikationspartnern mitzuteilen, wenn ein Dienst zu einem anderen Knoten

migriert ist.

Diese Arbeit stellt ein Modell vor, das es erlaubt, Prozesse die LibRIPC nutzen mittels frei wählbarer Dienstnamen zu identifizieren. Desweiteren stellt diese Arbeit einen Mechanismus vor, der diese Dienstnamen mittels eines zentralen Verzeichnisses im Netzwerk sicher zu Adressen auflöst. Ergebnis einer Namensauflösung ist ein Capability-Objekt, welches wir verwenden um Prozesse und Dienste zu adressieren. Indem wir Kommunikationspartner benachrichtigen wenn ein Dienst migriert, ermöglichen wir betroffenen LibRIPC-Instanzen, ihre Capabilities zu aktualisieren. Dadurch, dass LibRIPC Capabilities aktualisiert, stellen wir sicher, dass Prozesse weiterhin das selbe Objekt verwenden können um einen Dienst zu kontaktieren, unabhängig davon, ob dieser Dienst migriert wurde.

Wir beschreiben, wie wir einen Prototypen unseres Modells mittels Apache ZooKeeper [3] implementierten. Evaluation dieses Prototyps ergab, dass wir im Vergleich zu LibRIPCs aktuellem Auflösungsmechanismus das 35-fache der Zeit benötigen um eine Namensauflösung durchzuführen. Dennoch ermöglicht unser Prototyp das Senden von Nachrichten an über Capabilities adressierte Kommunikationspartner ohne Mehraufwand.

Contents

Abstract **v**

Deutsche Zusammenfassung **vii**

Contents **1**

1 Introduction **3**

2 Background **5**
2.1 LibRIPC 5
2.1.1 Broadcast Overhead 6
2.1.2 Security Issues 7
2.1.3 Stale Cache Entries 8
2.2 Apache ZooKeeper 9

3 Design **11**
3.1 Dynamic Service-Based Architecture 12
3.2 Capability-Based Addressing 13
3.3 User-Chosen Service Names 14
3.4 Resolution Service 14
3.4.1 Request Types 15
3.4.2 Data Persistence 16
3.4.3 Design Freedoms 17
3.5 Security Considerations 17
3.6 Service Migration 18
3.7 Future Work 19

4 Implementation **21**
4.1 ZooKeeper Znodes 22
4.2 Capabilities 22
4.3 ZooKeeper Watches 23

4.4 Naming Scheme API 23
4.4.1 ZooKeeper Connection 23
4.4.2 Capability Persistence 24
4.4.3 Service Registration 24
4.4.4 Service Migration 26
4.4.5 Service Lookup 28
4.5 Future Work 32

5 Evaluation **33**
5.1 Short Message Send Performance 33
5.2 Lookup Performance 35

6 Conclusion **37**

Bibliography **39**

Chapter 1

Introduction

Future cloud scenarios are expected to be realized in a fashion similar to current high-performance computers [2]. Instead of using few powerful nodes that are connected by a simple network technology, these applications are expected to consist of low cost nodes connected by a network that provides features like user-level I/O or remote DMA [4]. To make good use of these features, current high-performance network architectures require communication to be asynchronous and message-based [4]. Unfortunately, current operating systems present a socket interface for remote inter-process communication. Sockets offer only stream-based communication. Also, applications using sockets identify processes by their physical location in the network. In future cloud networks, hardware failure of individual hosts is expected to be more common than in current architectures [4]. Services hosted on these nodes can be restarted on different hosts, resulting in a change in their address. Furthermore, some applications might migrate processes between different hosts for load balancing. Without dynamic reconfiguration on side of the applications, processes in the network can not send messages to that service anymore.

LibRIPC [4] is an inter-process communication library designed for future cloud scenarios. It provides a message-based interface and improves the performance of cloud applications by taking advantage of user-level I/O and remote DMA. Instead of using physical network addresses, libRIPC uses hardware-agnostic service IDs to identify processes. If one process wants to send a message to another process, it has to know the service ID of the receiver. LibRIPC then resolves that service ID to the hardware-dependent network address of the receiver by sending a message containing that service ID to all processes in the network. After the process with that service ID has answered with its network address, libRIPC sends the message to that process. However, this approach has several disadvantages. First, service IDs are mere integers. They are non-descriptive and hard for humans to remember. Secondly, since resolution requests are sent

via broadcast, every process has to deal with every resolution request from any other process in the network. Broadcasting resolution requests creates communication overhead that a resolver based on unicasts could avoid. Thirdly, similar to ARP spoofing [7], any process could maliciously respond to a broadcast and try to impersonate the service that is to be looked up. Since libRIPC is designed for networks of mutually untrusted processes [4], its current resolver poses a security risk. Lastly, when a process migrates to a different host, communication partners continue to send their messages to the now invalid address. Each communication partner has to detect this situation and explicity send a broadcast for the process' new network address.

This thesis presents a naming scheme for libRIPC that tries to solve these issues. For users of libRIPC to be able to address processes in a descriptive manner, we decided to use strings instead of integers. We call these strings service names. A service name can only be resolved successfully if any process previously registered a service with that name at a distributed, central infrastructure. When a process wants to communicate with another process, libRIPC resolves the name describing the receiver to the receiver's actual network address. To avoid communication overhead, this request is sent as unicast to the central infrastructure in the network. The naming layer then returns a capability object which libRIPC uses to send messages to that process. Capabilities can be re-used without further cost, thus avoiding lookup overhead. Similarly, when a process has successfully registered a service, the naming layer returns a capability object which the process can use to update the network address of that service in the central infrastructure. Updating the network address of a service is not possible without the capability returned by the call to register that service, thus preventing security issues like man-in-the-middle attacks. When a service changes its network address, for example due to migration to a different host, our naming scheme updates the corresponding capabilities of all clients communicating with that service, and notifies the processes posessing those capabilities.

In Chapter 2, we present the background of this thesis. In Chapter 3, we describe the design of this naming scheme in detail, including process addressing, service identification, infrastructure required, security considerations, and handling of service migration, To test our design, we implemented a prototype using Apache ZooKeeper [3] as central network-internal infrastructure. We describe the prototype's implementation in Chapter 4. We evaluate the performance of this prototype and compare it against the current resolver of libRIPC in Chapter 5. Chapter 6 presents a conclusion.

Chapter 2

Background

2.1 LibRIPC

Future cloud scenarios are expected to be realized in a fashion similar to current high-performance computers [2]. Instead of using few powerful nodes that are connected by a simple network technology, they are expected to consist of low cost nodes connected by a network that provides features like user-level I/O or remote DMA [4]. LibRIPC [4] is an inter-process communication library designed for such cloud scenarios. It provides a message-based interface that maps well to the semantics of the underlying network hardware. LibRIPC presents an abstract interface, allowing users to design their applications independently of the actual network hardware used. Its message sending functions take advantage of user-level I/O and remote DMA to improve application performance.

LibRIPC uses hardware-agnostic service IDs to identify processes instead of physical network addresses. If one process wants to send a message to another process, it has to know the service ID of the receiver. LibRIPC first resolves that service ID to the hardware-dependent network address of the receiver. When the service ID has been resolved, libRIPC sends the message to the network address that it obtained by resolving the service ID. [4] This approach has several benefits. Service IDs free processes using libRIPC from dealing with actual network addresses and thus from dealing with the actual network hardware. There is no need for users of libRIPC to know or care about the physical location of communication partners. Applications in cloud networks can be distributed among computation nodes. When a host fails, its processes can be restarted on different hosts. In future cloud networks, hardware failure is expected to be more common [4]. When a process migrates from one node to another, resolution of its service ID simply results in a different address. Additional services can be added to an already existing network simply by using new service IDs, existing services

can be removed by stopping the corresponding processes [4]. However, if all the services in the network are described merely by integers, humans will have a hard time remembering which integer describes which service. A service ID does not describe the process it identifies.

To resolve service IDs to hardware addresses, libRIPC sends a broadcast containing the service ID to be resolved. A responder thread in each libRIPC instance then receives the broadcast and, if it is the intended receiver, answers with a unicast containing the network address of the process. This approach creates several problems. First, every process is involved in every resolution request, which creates communication overhead. If the broadcasted service ID does not belong to a certain process, that process can either discard the resolution request or use the source service ID the sender attached to fill its local resolver cache. We discuss this overhead in section 2.1.1. Secondly, every response to a resolution request is trusted. Since every process in the network receives the broadcast, every process could respond with a resolution response. Thus, malicious processes have the chance to perform a man-in-the-middle attack. We discuss security issues of the current resolver in section 2.1.2. Thirdly, lookup always has to be explicit. Clients of a service won't be notified when the service, for example, migrates to another host. Since libRIPC caches previously resolved service IDs, the cache will contain invalid data in this case. Section 2.1.3 discusses the problem of stale cache entries.

2.1.1 Broadcast Overhead

When sending a message to a service ID, the current resolver in libRIPC uses a broadcast to request the network address of the corresponding process. This request is received by every libRIPC instance in the network. Upon reception of a resolution request, a libRIPC-internal thread decides whether to ignore this request, use the attached sender service ID and sender network address to fill its local resolver cache, and/or answer the request with its own hardware address.

Since every process is involved, this approach creates communication overhead. To mitigate the overhead, the current implementation inserts the sender service ID and sender hardware address into its local resolver cache, provided that this service ID is not yet contained in it. Thus, a lookup can be avoided if the owner of the service ID to be looked up has performed a broadcast while the libRIPC instance that is about to perform a resolution was already running.

However, the resolver cache is not of much use in certain cases. Consider the following scenario. A network of many client processes and few services; clients want to use several of these services. The client processes are periodically stopped and restarted. For each initial resolution request by a new or restarted client, all other clients and services will receive the broadcast message. Furthermore, if the

clients are communicating with services only, caching the sender service ID of resolution requests sent by other clients provides no benefit.

Responses to a resolution request are sent as a unicast. The libRIPC instance corresponding to a the service ID that has been queried sends its response only to the sender of that broadcast. Thus, when a service is migrated, all of its clients will have to resolve its service ID again, which results in one broadcast for every client.

2.1.2 Security Issues

LibRIPC does not use any form of authentication. It follows that every response to a broadcasted resolution request is trusted and that every process in the network can respond with such a response. Similar to ARP spoofing [7], attackers can perform man-in-the-middle attacks. A malicious process can answer resolution requests with its own network address, resulting in an impersonation or man-in-the-middle attack. If the attacker's response is received first, it will be accepted by the sender of the resolution request. After the resolution request has been handled, the sender of the resolution request will send its messages to the attacker.

If an attacker has successfully performed an impersonation and the rightful owner of that service ID is online, the owner will send its hardware address to the sender of the resolution request as well. The sender of the resolution request will thus receive another response to the same broadcast in this case. Thus, receiving two answers to a single broadcast indicates that one of these two responses could have been malicious. The current scheme could be extended to detect this situation and drop both addresses. While this modification could be used to detect and stop attacks after they have occurred, a malicious process would then be able to perform a denial of service for every lookup. The attacker could send a response to any resolution request. When the sender of the resolution request receives the second response, he would invalidate the entry for the service ID in the cache. Thus, this entry would be invalidated regardless of whether the answer of the attacker or the one of the victim was received first. Before the sender of the original resolution request could continue sending messages to that service ID, another resolution would have to be performed, which would result in the same problem.

Since libRIPC uses a resolver cache, impersonation or man-in-the-middle attacks are possible in another way. An attacker need not to wait until a resolution request for its victim is sent, he can also actively inject invalid information into the network without having to wait for an event. For this purpose, the attacker attaches the victim's service ID to an arbitrary resolution request. All processes in the network will receive this broadcast and insert the mapping of the victim's

service ID to the attacker's network address into their resolver caches if their respective cache does not yet contain an entry for that service ID.

Since libRIPC considers processes to be mutually distrusting [4] and there is no trusted instance on the network, all simple dynamic strategies to solve these problems require the victim to be online when using the current broadcast-based scheme. As soon as a process is not running, any process could take over its service ID. Without additional precautions taken by the applications, such attacks on the current resolver can not be noticed.

2.1.3 Stale Cache Entries

Due to reasons like hardware failure or load balancing, processes might be migrated between hosts. In some scenarios, process migrations might occur frequently. Furthermore, processes might crash and take some time to be restarted, or be restarted on a random host. In the current resolver, clients of a service won't be notified when the service is migrating to another host, when it is shutting down temporarily or when it has restarted on another host. Thus, the resolver caches of the service's clients will contain a stale entry in these cases. Since libRIPC uses asynchronous, unreliable messages [4], sending messages to a service whose cache entry is stale results in a silent failure. To detect a failure in sending a message, applications need to implement their own error handling.

If a process implements, for example, timeouts and detected a stale cache entry, it has to perform another resolution of the same service ID. Resolution will result in a different physical address, or no answer at all if that service is down. A service might be down temporarily during migration. To handle temporary downtimes of the service, processes have to try to resolve the service ID periodically. Each attempt requires a broadcast to be performed, which results in "polling" for the new network address by broadcast messages until an answer is received. Having to rely on busy waiting via broadcasts creates even larger overhead than using broadcasts for regular resolution alone.

One possible solution against busy waiting would be to allow applications to discard entries from libRIPC's resolver cache. When an application recognizes a service as being down, it would clear the corresponding cache entry. When migration is completed, the service would broadcast its new address, either as a special message or as an attempt to resolve an arbitrary service ID. Since the client then would not have an entry in its resolver cache for that service ID, it would cache the new network address of the service that has just been restarted. Additionally, libRIPC could and should execute a user callback function to notify the application of this event. Upon receiving this event, the application would continue its normal operation. However, the client's libRIPC instance would behave the same way if an attacker sends such a broadcast with the goal to impersonate that

service. Also, broadcasting the new network address would notify every process in network. Most of these processes would never contact the service. Thus, the workaround would solve neither of the two problems we discussed in the previous subsections.

Thus, if applications do not build additional mechanisms on top of libRIPC, they will suffer from stale cache entries and silent failures when sending messages. Detecting and handling this situation inflicts high management overhead on applications, further communication overhead on the network and additional latency while waiting for timeouts. We therefore conclude that libRIPC needs a mechanism for services to notify all communication partners about a change in its power state or physical location.

2.2 Apache ZooKeeper

ZooKeeper [3] provides mechanisms to coordinate distributed processes. As opposed to other coordination mechanisms, ZooKeeper does not provide synchronization primitives but rather exposes an API that lets developers create their own primitives. ZooKeeper does not provide any locking mechanisms. It rather presents information created by different processes as a file system like tree hierarchy called data tree. Nodes in this tree are called znodes. Modification of the data tree is done in a wait-free manner: Clients can send several asynchronous read or write requests without having to wait for their completion. Nodes in the data tree can at the same time contain data and have children. [3]

A ZooKeeper ensemble consists of at least one server. Each ZooKeeper server instance has a local copy of the data tree and communicates write requests back to the other servers. ZooKeeper clients connect to an arbitrary server of an ensemble to establish a session. Once a client has established a session, several properties hold for this session: First, the client can connect to an any other ZooKeeper server in that ensemble and still continue using the same session, and thus, the client will see all changes it made to the data tree while being connected to a different service. Secondly, multiple requests of that client are processed in the order they were sent. Thirdly, any read request sent by this client can and will be satisfied by the ZooKeeper instance the client is currently connected to. Lastly, any change to the data tree will be sent to all other servers of the ensemble and thus be visible by all other clients, regardless of which server they are connected to. [3,5]

When issuing any kind of read request, clients can set a so-called "watch" by passing a boolean flag. Clients that have set a watch on a certain znode when reading its contents or children will be notified when any client changes the data that was queried. Upon receiving the notification, the client can issue another

read request to get the latest version of the data, again with the option to request notification in case of changes. [3, 6]

While supporting fast writes, ZooKeeper was designed for scenarios in which frequent reads are the common case [3].

Chapter 3

Design

The naming scheme presented in this thesis tries to improve libRIPC's current naming and resolution mechanism. Thus, we want to maintain its advantageous properties.

One benefit of libRIPC's current approach is its flexibility. Processes can be added to and removed from an existing network at any time. Our naming scheme maintains this property and extends it with the differentiation between regular processes and services. Thus, libRIPC realizes a *dynamic service-based architecture* for networks, as we describe in section 3.1.

Another benefit of the current version of libRIPC is that processes can address communication partners independently of physical locations. Using *capabilities* instead of service IDs allows us to keep addressing transparent. Users can treat capability objects similarly to service IDs but do not have to know their contents. We describe how we designed capabilities in section 3.2.

Applications using libRIPC identify services by *user-chosen service names*. A service's name is chosen by the user when he registers that service, thus allowing services to be identified in a descriptive manner. As we describe in section 3.3, it is possible to modify the current resolution mechanism to support service names instead of service IDs. However, all of the other problems of the current resolver would remain unsolved. Thus, we designed a completely different resolution mechanism.

To avoid the overhead caused by broadcasts, processes send resolution requests via libRIPC to a central *resolution service* in the network. Lookup of a service name succeeds if some process previously registered a service with that name. To register a service, a process sends a service creation request via libRIPC to the resolution service. Section 3.4 describes that resolution service in detail.

To decrease lookup overhead even further, successful resolution of a service name returns a capability object. This object authorizes the process to send messages via libRIPC to that service, regardless of the service's physical location.

Capabilities also allow us to realize *security*: Upon creation of a service, libRIPC's naming layer again returns a capability. To publicly update a service's network address, a process has to send an address update request to the resolution service. The resolution service prevents unauthorized updates by ensuring that this request will only succeed if the process submitting the request possesses the capability that was returned on creation of that service. We describe our approach to security in section 3.5.

To be able to provide efficient location-transparent addressing even in the face of process migration, libRIPC instances have to be notified about changes in the network address of services they use. Section 3.6 presents our design of this notification.

There remains further work to be done on libRIPC and this naming scheme, which we discuss in section 3.7.

3.1 Dynamic Service-Based Architecture

The naming scheme presented in this thesis differentiates between a *process* and a *service*. Services are realized by processes. Each process can provide zero or more services.

A service is an abstract entity. While each service is provided by at most one process at any given time, there can be another process providing that service later. For example, when a process crashes, a different process of the same program can be created and start providing this service. As long as the protocol between the service and its clients has been designed with recoverability in mind, communication between clients and the new instance of the program can continue without problems. It is also possible to create different implementations (for example different patch-level versions) of one service and switch between them at random by stopping the current process and executing a different program.

Regular processes have no service name and thus can not be looked up. They can resolve service names and take part in communication. Like a service can be realized by multiple processes, each process can provide as many services as wanted.

Submitting resolution requests is possible for any process in the network. However, only services can be looked up. Even though regular processes can not be resolved, bi-directional communication is possible. Every message sent via libRIPC contains addressing information about the sender, allowing the receiver to respond to the message.

3.2 Capability-Based Addressing

To send messages to a process or service, we need some token that addresses a certain process. Using network addresses only would make the naming scheme inflexible, as communication would break off and could not be restored when the receiver is a service that has just migrated. While this form of addressing would be possible for regular processes because they cannot be looked up anyways, we decided to make the API of our naming scheme conistent. In other words, a user of libRIPC has to be able to send messages in a uniform way, regardless of whether the receiver is a regular processes or a registered service, and regardless of the current physical location of the receiver.

We chose to use capabilities for addressing communication partners. A capability contains the network address of the process it references. In case the capability references a service, the capability additionally contains the name of the service. In case the capability is obtained by a call to register a service, the capability additionally contains an access token required for authentication. We call the latter kind of capability a service management capability. However, the user of libRIPC does not need to know about the contents or type of a capability. The user can send messages to a certain process or service via libRIPC if, and only if, he supplies a capability object that references that process. When the receiver is a service, the sender does not need to care whether the service has migrated to a different host. By using an appropriate capability, libRIPC makes sure that the message will be sent to the endpoint providing that service, even if the endpoint has changed since the last message sent to that service by that process. Furthermore, capabilities are particularly suited for service management. By having the service creation functions return a highly-privileged capability for that individual service, we initially allow only the creating process to manage that service. By using capabilities for service management as well, our API addresses both processes and services solely via capabilities and is thus easy to learn and use.

It is up to the user to decide how capabilities are used. For example, a process can pass a capability to another process, allowing the receiver to contact the corresponding process, manage the corresponding service, or do both, depending on the actual capability.

Using capabilities seperates policy from mechanism. Internals of and implementation details concerning capabilities are managed by libRIPC. Neither does the user need to know the contents of capability objects nor how exactly they are accessed by libRIPC. However, it is up to the user to decide *how* they are used. Thus, capabilities allow for fine-grained, dynamic access control.

3.3 User-Chosen Service Names

To allow descriptive addressing of services, we want to extend libRIPC to support user-chosen names to identify services.

There exist several workarounds to add this feature on top of the current resolver. As a hack, one could use an additional central C or C++ header file using `enum` or `#define` to define service IDs as constants with "speaking names". This header and its corresponding library could also contain a mapping from those constants to the actual service names as strings. Since clients usually use only services that have been planned or designed before or during the time the client was developed, this header file would contain all services a client might want to use.

A cleaner solution would be to modify the current broadcast scheme to not broadcast an integer but a string of characters. This string is the name of the service to be resolved and could have a fixed or maximum length. The libRIPC instance in the corresponding service would, after receiving the broadcast, respond with the current hardware address of the process.

There might be other approaches that could be used to realize service names on top of the current resolver. However, all of the other problems we described would remain unsolved. Thus, we decided to design a completely different resolution mechanism.

For this purpose, services are registered at a central network-internal infrastructure. A service is registered by a process sending a service creation request and a string to that infrastructure. That string will be the name of the service after it is registered. While its contents are arbitrary, in practice, that string is likely to be hard-coded during development of service and client programs.

Service names identify services like services IDs do. However, as opposed to the current design, users of libRIPC will not pass them directly to libRIPC's message sending functions. Instead, they explicitly perform a resolution of a service name, which results in a capability. LibRIPC's message sending functions will only require such an object, they will not perform an implicit resolution of service names. This approach seperates the concerns of resolution and message sending, and allows for more flexibility in usage while maintainining the level of abstraction.

3.4 Resolution Service

To prevent the communication overhead of the current resolver, our naming scheme is designed in a way that communication involves interested processes only. Thus, libRIPC sends requests relevant for the addressing of services to only one *resolution service*. The tasks of the resolution service are maintaining a central directory

of services names and addresses, and answering requests concerning the address resolution of those services.

3.4.1 Request Types

There are four kinds of requests the resolution service must understand and handle properly: Requests to resolve a service name to the current network address of that service, requests to create a service with a certain name, requests to update the current network address of a certain service, and requests to delete a certain service.

Service Name Resolution Request

A resolution request contains a string to be looked up. When receiving a resolution request, the resolution service answers with the physical address of the service with that name, or a descriptive error number. To prevent communication overhead, this answer is sent only to the libRIPC instance that sent the request. The libRIPC instance then creates or updates a capability and returns it to the user.

Service Registration Request

Initially, the resolution service contains no entries. To perform an address resolution of a certain service, that service first has to be created.

For this reason, a process can send a service registration request. That request contains the service name. The service creation request is successful if there is not already a service with that name. When the request was successful, the service name is stored in the internal directory. Resolution of that service name is then possible globally. Also, the resolution service responds with an access token used for authentication. The libRIPC instance that sent this request uses this token to create a capability which it returns to the user. Thus, the process that created the service obtains a capability that can be used to update that service's network address via address update requests and to delete that service via deletion requests.

Service Address Update Request

To support process migration, a process providing a certain service has to be able to publicly change the network address of that service. Since this address is stored by the resolution service, the resolution service has to provide functionality for changing service network addresses. A service address update request contains the name of the service to be updated and the new address of that service.

The network address of a service can be changed by all processes that possess the corresponding capability that was returned as response to the registration request of that service. Note that the process sending the update request does not need to be the process currently providing that service. For example, scenarios are possible in which a certain process is responsible for migrating other process. We call the process responsible for migrating other processes migration service. If this migration service possesses service management capabilities for the processes to be managed, it could update the network addresses of those processes after having moved them to a different host, thereby making migration—and, thus, physical location—transparent for the process that is migrated.

Service Deletion Request

Administrators can remove a service by shutting down the process or processes providing that service. If the administrator removes all client processes of that service as well, all that remains is the entry of the service name in the data storage of the resolution service.

In some cases, administrators might want to remove that entry as well, allowing processes to re-register that service name. For that reason, any process that holds a capability that was returned by the service registration request can delete that service.

Note that once that service has been deleted, any process in the network is free to register another service under that name. If there are clients of the old service left, a malicious process can take advantage of the situation by re-registering that service name and impersonate the service that has just been deleted.

3.4.2 Data Persistence

The resolution service has to store information about registered services. This information consists of service names and the current network address of each service. To support service management capabilities in this distributed setting, the resolution service also has to store a hash of a large and random identifier for each service. Posessing this access token allows processes to change the corresponding service's network address or to delete that service.

The resolution service has to store this information in a persistent manner, so that the network survives restarts of the resolution service. Thus, when a service has been created, that service exists until it is explicitly deleted. Data associated with that service will only change if a user sends either an address update request or a service deletion request using the corresponding management capability.

3.4.3 Design Freedoms

Our design leaves many freedoms for the implementation of the resolution service.

For example, we do not define how or where the information has to be stored. storage could, for example, be handled by a seperate process. Also, there could be not just one but several resolver processes in the network. These processes could employ some form of replication or distribution.

However, a libRIPC instance will always communicate with only one process of this set. Using two or more resolver processes is possible as long as the information stored by those processes is consistent. That is, if a request to create a service was sent to one process of the resolution service and succeeded, there must be no conflicts if another process has just created a service with the same name via a different process of the resolution service. Thus, service creation must be propagated to all processes of the resolution service without conflicts. This propagation must happen in a reasonable amount of time, so that all processes in the network can resolve the service name and use the service, regardless of which process of the resolution service they are connected to. Changes in the network address of a service have to be propagated as well and are more time-critical. When a service continues being available on a different endpoint, its clients should receive information about the change in the services network address as soon as possible.

Furthermore, when a libRIPC instance has created a service, the processes of the resolution service have to make sure that this libRIPC instance can reconnect to different server and still be able to manage the newly created service.

3.5 Security Considerations

To guarantee secure operation of cloud applications using libRIPC, we have to make sure that messages are sent to the intended receiver. Attackers must not be able to redirect the flow of communication.

By using a central, trusted resolution service and knowing its network address in advance, there is no possibility for regular processes in the network to attack the message routing between libRIPC instances and the resolution service. Our naming scheme thus will not communicate with an attacker during lookup.

To actually prevent man-in-the-middle attacks against communication between processes, we have to go further and guarantee that the resolution service returns authenticated information only. Thus, the resolution service must only accept update or delete requests sent by authorized processes. For this reason, a successful service creation request returns a capability. Updating that service's address or deleting that service can only be done by processes that possess this capability. Also, when a service has been created, processes have to be able to rely on the

fact that a simultaneous service creation request for a service with the same name will not succeed and grant the same rights. When a process tries to register a service, that request fails if a service with that name already exists. If one request succeeded, all requests to create a service with an identical name will fail, regardless of the order in which these requests were transmitted and regardless of which process of the resolution service these requests were sent to.

It is important that the resolution service remembers services from when they have been created until they are explicitly deleted. If the resolution service would only keep track of currently running services, an attacker could wait until a service goes offline temporarily, for example due to a hardware crash, and then register that service name again with its own network address. Performing man-in-the-middle attacks would be easy.

3.6 Service Migration

Due to reasons like hardware failure or load balancing, processes might migrate between hosts. In some scenarios, process migrations might occur frequently. Furthermore, processes might crash and take some time to restart, or be restarted on a random host. LibRIPC abstracts from hardware details like a service's physical location in the network. It follows that migration of a certain service from one host to another should not have any noticable impact on clients currently communicating with that service. It should be possible for communication to continue without requiring that users of libRIPC intervene and handle this situation. Thus, to be able to realize reliable, persistent communication, we have to consider these scenarios.

We designed our naming scheme in a way to automatically perform notifications when a service continues being provided at a different address. Similar to the case against broadcasted resolution requests we made, we decided against broadcasting these notifications. Instead, notifications are sent only to those clients that are currently interested in an up-to-date network address of the service whose physical location has changed. To indicate whether a user of libRIPC is interested in notifications about changes in a service's physical location, libRIPC passes a boolean flag when requesting resolution of a service name to that service's currently effective physical location. A client is deemed to be interested in notifications if no previous notification has been sent to this client since it it last indicated its interest.

Notifications do not contain the new address of the service. They merely inform a client about the name of the service which triggered the event, and whether that service has just been created, deleted or whether its address has simply changed. After receiving a notification, libRIPC can decide whether or not to

perform another resolution of the service name contained within the notification message. Until this libRIPC instance performs another resolution of that service name with the flag set to indicate interest in notifications, the corresponding process will be regarded as not interested in that service's current physical location and thus not receive further notifications. By following this approach, we can prevent superflous notifications, and thus, communication overhead. Until the client receives a response to a resolution request, it is aware that its local information is outdated, regardless of how many times the service's address has changed since the client received the notification. Once the resolution service has started sending its answer, the client will again be notified if that service changes its physical location, provided the client indicated further interest. Thus, there are two reasons for clients not receiving information about an intermediate state: disinterest and slow operation. In both cases, there is no need to send another notification to that client. As soon as that client performs a resolution, it will receive the latest information.

To guarantee secure operation of the network, we have to ensure that information about service addresses is authenticated. Notifications do not include any addresses but rather require the affected libRIPC instance to perform another lookup. Since the information returned by these requests is authenticated, the notification feature of our naming scheme can not be misused by attackers to inject invalid addresses.

Notifications are sent upon change in the service's physical location in the network. Thus, clients are not only notified when a service continues being available on a different host, but also when that service stops being available at all.

3.7 Future Work

Currently, every process can register any service name that is not yet registered. Malicious processes could try to exhaust available service names or register service names that are similar to names of currently existing trusted services. For this reason, scenarios might require that creation of services is only allowed after these requests have been checked by a trusted administrator.

There exists a race condition between message sending and address change notification. Consider the scenario that a process has looked up a service. Now, that process sends messages to that service while simultaneously that service migrates to a different host. After the service has logged out and stopped being available under its old address, the process might not yet have received the notification about the change in the power state of the service. Thus, some of the messages the process sends will not reach their destination but result in silent failures. The process will only become aware of that fact when it received the notification. At

that point, however, libRIPC provides no means of reporting which or how many message sends failed.

Similarly, when a process providing a service crashes or fails to perform a logout, the service will still be registered as online and reachable under a certain address. Other processes could successfully perform a lookup but would repeatedly fail to send messages. As a consequence, they would have to implement additional logic to deal with invalid responses of the resolution service. Additionally, if that service is down for a long time, attackers could take note of it and try to start a malicious process that is reachable under the hardware address of the defunct service.

Both of these problems could be solved by using "heart beats" or "pings". When a process has changed the network address of a service, libRIPC periodically sends a request to the resolution service. if the resolution service does not receive such a request withing a certain amount of time, it assumes the corresponding process is dead and will clear the network address of that service. Also, any process can sent a request to the resolution service indicating that a certain service appears to be down. The resolution service then sends a message to that service's libRIPC instance to check whether it is dead. If so, the resolution service will clear the network address of that service to reflect that the service is offline.

Chapter 4

Implementation

To test the design of our naming scheme, we implemented a prototype. Implementation of our naming scheme consists of two major components: extension of libRIPC and the network-internal resolution service.

Apache ZooKeeper provides coordination mechanisms for distributed processes. Our research proved it to be suited for most tasks required from our resolution service. In the following, we show how we configured and used ZooKeeper to realize our prototype.

Apache ZooKeeper stores data in a filesystem-like hierarchy called data tree. The data tree consists of so-called znodes. Each znode is discribed by a unique path and can contain user-chosen data. [3] ZooKeeper controls access to znodes by using Access Control Lists (ACLs) [6]. In section 4.1 we show how we store service names and their current network address in ZooKeeper's data tree and how we control access to znodes.

We use ZooKeeper to realize central storage of service information in the network. For process-local storage, we designed capabilities as images of znodes. Our capabilities contain the name of the corresponding service, its network address, and all information required to authenticate a client with ZooKeeper. Thus, they allow for transparent addressing and realize authorization. We describe the design of our capabilities in detail in section 4.2.

A process starts providing a service by writing its network address into the corresponding znode. Persistent addressing of services that have migrated requires updating process-local capabilities with the latest information contained within znodes. Thus, to support both service migration and persistent addressing of migrated services, we have to synchronize capabilities and znodes. Section 4.3 describes how we use ZooKeeper's notification mechanism to update locally cached information about services. In section 4.4, we present the API of our naming scheme and how we use ZooKeeper to update network-global service information stored within znodes.

4.1 ZooKeeper Znodes

Apache ZooKeeper stores data in a filesystem-like hierarchy called data tree. Nodes in this tree are called znodes. The structure of the data tree can be modified dynamically by ZooKeeper clients by creating or deleting znodes. The data tree is persistent [6]: ZooKeeper periodically creates snapshots and stores them on the hard disk.

Znodes are referenced by paths similar to file paths in filesystems. As opposed to traditional file systems, each znode can both have children and store information. ZooKeeper clients can read data from and write data into znodes. ZooKeeper controls access to znodes by using Access Control Lists (ACLs). An ACL is a list of entries, each consisting of an identifier and a set of permissions [8]. Identifiers consist of a string describing the authentication scheme to be used and a string containing credentials required to verify authentication attempts of clients [8].

Since our naming scheme requires network addresses to be looked up by service names, we use service names as keys for ZooKeeper. Thus, for each service, there is a znode that is referenced by the name of the service. That znode contains the corresponding network address. For simplicity, we use the service name without any modifications to identify the corresponding znode.

To group all services created and managed via libRIPC and to avoid interference with other processes using ZooKeeper, we create znodes as children of a libRIPC base znode. Thus, given a service named `S`, the ZooKeeper path of the znode of that service is `/libRIPC/S`.

4.2 Capabilities

We implemented capabilities as local caches of the network address stored in znodes and as token that authorizes processes to change this data in ZooKeeper.

A capability is a data structure that contains the name of the service it references, a cache of its network address, and authorization information. Since capabilities contain service names, they also identify the corresponding znode.

When creating a new capability to be used to register a service, libRIPC generates a large and random access token. This token is stored in the capability and used to create credentials for the ACL of that znode. To modify that znode after creation, the client has to authenticate itself at the ZooKeeper server it is connected to. For this purpose, the client needs to supply the correct credentials. Since the access token is stored within the capability, as long as a client possesses the capability, it can modify the corresponding znode. While all capability objects are presented in a uniform manner to the user, in the following, we call such a capability a service management capability.

Capabilities are stored in user-level. Thus, they are forgeable. However, write access to znodes is controlled by ACLs in ZooKeeper. A write is only permitted if the client transmits the correct access token. Since the token is large and random, we prevent unauthorized modification of znodes, and thus unauthorized changes in the network addresses of services. In fact, due to the flat and self-contained structure of capabilities, they can easily be serialized for persistent storage or to be sent to other processes.

4.3 ZooKeeper Watches

When reading a znode, ZooKeeper clients can indicate their interest in being notified if the data of a znode changes by passing a boolean flag, called "watch". While ZooKeeper provides no means to remove a watch that has been set [1], each watch will only be triggered once.

When a znode is written, ZooKeeper will notify all clients that have set a watch for this event and znode. This notification will contain the path of the znode that was changed. It will not contain the new data of the znode. To obtain that data, ZooKeeper clients have to read that znode manually after receiving the notification [3].

When reading a znode, the ZooKeeper client library not only allows user to set a watch, but also to pass a pointer to a "watcher context". When the ZooKeeper client receives a notification about the change of a certain znode, the client library will call the watch event handler once for each call of the read function that passed a watcher context. The watcher context that the user passed to the read function of the ZooKeeper client library will be passed to watch event handler as parameter.

4.4 Naming Scheme API

In the following, we describe how we implemented our API by using the ZooKeeper client API. Our API is coarsely divided into connection management (section 4.4.1), capability management (section 4.4.2), service registration (section 4.4.3), migration (section 4.4.4), and lookup (section 4.4.5).

4.4.1 ZooKeeper Connection

Before a process can register or look up services, libRIPC needs to connect to ZooKeeper. While our prototype connects to ZooKeeper implicitly, the user needs to specify a list of ZooKeeper servers. LibRIPC stores this string internally and

adds a globally constant "chroot" suffix. This string will be passed to the initialization function of ZooKeeper's client library when connecting. Similar to the Unix command `chroot`, all path names to znodes sent ZooKeeper by using this connection will be resolved with the znode referenced by the "chroot" suffix as "root directory" [6]. Thus, libRIPC won't accidentally read or write znodes that are used by other processes accessing the same ZooKeeper ensemble, as long as those processes do not access libRIPC's base directory or its children.

When a function of our API is called that requires a call to ZooKeeper to take place, our prototype first checks whether it is currently connected to ZooKeeper and, if not, instructs the ZooKeeper client library to establish a connection by using the previously stored address string. The ZooKeeper client will then connect to an arbitrary host of this list.

4.4.2 Capability Persistence

Since capabilities are stored in local main memory, a process that possess a certain capability will lose that capability if it crashes. After that process has restarted, services that were registered in ZooKeeper before will still exist. If that process was the sole process providing a certain service, all management capabilities for that service will be lost. Data required for authentication to modify the corresponding znode was only contained within that capability. Due to the crash, no process in the network can authenticate itself as "owner" of the corresponding znode. Thus, to prevent loss of authorization for service management, processes have to be able to persist capabilities locally. For this reason, we designed `capability_serialize` and `capability_deserialize`. The user of libRIPC can call those functions to store an individual capability in an individual file, or respectively to read an individual capability from an individual file.

4.4.3 Service Registration

Before a certain service name can be resolved successfully, a service with that name has to be created. A call to `service_create` registers a service at ZooKeeper by creating a znode with the name of the service.

LibRIPC ensures that after a successful call to `service_create`, the process possesses a capability that can be used to manage that service. This capability grants the process that possesses it permission to change the service's network address, to send messages to it, and to receive messages sent to that service.

Note that attempts to serialize that capability could fail, for example if there is no space left on the filesystem. Even if the process immediately calls `capability_serialize` after it has registered the service, the process still might

crash before it has stored this capability on persistent storage. As a result, ZooKeeper would contain a znode but there would be no process in the network that is authorized to write into that znode. To prevent this situation from happening, we designed `service_create` to expect a capability as parameter instead of a service name. This capability references the process that is to be registered as service. A process can obtain a capability referencing itself by calling `capability_create`. `capability_create`, on the other hand, expects a name as parameter that identifies the process. `service_create` will name the service after the process identified by the capability passed as parameter. Thus, processes can decide on a name, create a capability referencing themselves, serialize this capability, and then try to register a service. Even in case the process crashes immediately after registering the service and another instance of that program is started, that instance can deserialize the capability and continue providing that service.

To create a service, `service_create` ultimately creates a znode in ZooKeeper. To create a znode, ZooKeeper requires a path name, an ACL, and the data to store in the znode to be created.

Znode Path Name

We want to store all znodes representing services as children of the libRIPC base znode. Our connection to ZooKeeper already specifies this znode as "chroot" base znode. Since all paths in ZooKeeper have to be absolute, we create the znode path by simply prefixing the service name with a "/".

Znode ACL

In our setting, we have to consider two roles when creating the ACL for a znode. First, every process is allowed to look up any service. Thus, every ZooKeeper client has to be authorized to read the znode to be created. Secondly, modification of a znode is only allowed for processes possessing the corresponding management capability. Thus, the ACL has to contain an entry that authorizes the client creating the znode to modify the znode afterwards. Other clients must not be authorized to do so, unless this permission has been passed on. Thus, the ACL we pass to ZooKeeper consists of two entries: The first gives read access to any client, the second gives read and write accss to all clients that possess the management capability.

We decided to use the ZooKeeper-builtin "digest" scheme for determining whether a client possesses the management capability. The digest scheme expects credentials in ACLs to be made up of a user name, followed by a colon, followed by a base64-encoded sha1 hash of user name and password sepearated by a colon. In pseudo-code: `"username:" + base64(sha1("username:pass`

`word"))`. We use the service name as user name. During creation of the capability, a large and random access token has been generated and stored within the capability. We supply this token as password.

To modify that znode after its creation, the ZooKeeper client has to authenticate itself at the ZooKeeper server it is connected to. The client authenticates itself by providing user name and password. Thus, the client has to transmit service name and access token. Both service name and access token are stored within the management capability. Thus, as long as a client possesses the management capability, it can modify the corresponding znode. Since the access token is practically unforgeable, we prevent unauthorized modification of znodes, and thus unauthorized changes in the reporting of service network addresses.

Znode Data

`service_create` expects a capability as parameter. The process referenced by this capability is registered as service. Thus, the znode we create initially contains the network address stored within that capability.

4.4.4 Service Migration

A service migrates to a different endpoint if the process that currently provides that service is shut down and another process, probably of a different program, starts providing that service under a different network address, probably on a different host.

LibRIPC instances of clients of that service have to be notified about the fact that the service stops being available under its current network address, and, when another process has taken over, about the new address of the service. For this purpose, we designed the functions `service_logout` and `service_login`, respectively. Both expect the user to supply a management capability and implicitly authenticate the ZooKeeper client at the server.

Authentication

We change the network address of a service by changing the data contained in the corresponding znodes. Since writes to znodes are only possible for authenticated clients, libRIPC checks whether it has performed this authentication during the current connection to ZooKeeper. If libRIPC did not yet authenticate itself as a client possessing the right to write this znode, it will add an authentication to the current connection.

Authentication requires sending the identifier for the scheme to be used, as well as the credentials to be used for authentication. Contents of the credentials

depend on the scheme to be used. LibRIPC uses the “digest” scheme. In this case, libRIPC has to pass a string consisting of user name and a password sepearated by colon to the ZooKeeper client library [6]. We created the znode’s ACL with service name as user name and a randomly generated access token as password. Since both service name and access token are stored in the capability, all information required for authentication is contained within it. LibRIPC uses this data to create the credentials required by the “digest” scheme. It sends these credentials to ZooKeeper to add authentication information to the current connection. Afterwards, this ZooKeeper client has write access to that znode.

Service Logout

To notify clients of a service about the fact that the service stops being online, the process currently providing the service has to pass the management capability to `service_logout`.

When `service_logout` is called, libRIPC first checks whether its ZooKeeper client is currently connected to a ZooKeeper server and creates a connection, if necessary. Since writes to znodes are only possible for certain authenticated clients, libRIPC checks whether it has authenticated itself as a client possessing the right to write this znode. If libRIPC has not yet performed this authentication for the current connection to ZooKeeper, it will do so.

To mark the service as offline, libRIPC clears the data contained in the corresponding znode by doing a standard ZooKeeper write with length 0. Writing data to a znode triggers notifications in ZooKeeper. Thus, when libRIPC has cleared the address of the service, ZooKeeper notifies all ZooKeeper clients that have set a watch. The ZooKeeper client executes the event handler of the libRIPC instance it belongs to. Each libRIPC instance then updates its capability for that service.

Service Login

To start providing a service, a process passes the management capability to `service_login`. LibRIPC determines the current hardware address of the process. Again, libRIPC tries to ensure that it is connected to ZooKeeper and authorized to change the znode referenced by that capability. Afterwards, libRIPC writes the network address it has determined into the znode references by the capability. Writing this address into the znode results in the same event as clearing the contents of that znode. Thus, notifications will be triggered and handled in the same manner.

4.4.5 Service Lookup

The primary use case of the naming API is service name resolution. Our API supports both automatic updates and manual lookup. While the latter performs service name resolution only once, the former variant additionally requests notification in case the service to be looked up changes its physical location or ceases to exist.

LibRIPC resolves a service name `S` by trying to read the znode located at `/libRIPC/S`. Since the base znode `/libRIPC` has been set as chroot prefix during login to ZooKeeper, libRIPC actually passes the path `/S` to the client library of ZooKeeper. If that znode exists, ZooKeeper responds with the data contained in the znode and the ZooKeeper client library writes this data in a buffer supplied by libRIPC. The ZooKeeper client library responds with a result code indicating success or the type of error that occured. If reading the znode was successful, libRIPC stores this network address in a capability. This capability will then be returned to the user. It can only be used to send messages to the network address currently providing that service. It cannot be used to receive messages sent to that service, manage that service's network address, or delete it. Lookup fails if no service with the specified name is currently registered. That is, if no znode with that name exists. Lookup succeeds even if the service is currently offline. In that case, the corresponding znode does not contain data. LibRIPC will return a capability for this znode nevertheless. Combined with automatic updates, this feature allows users of libRIPC to wait for startup of a certain service.

It is possible to resolve a single service name several times and thus obtain multiple capabilities for the same service. Users can decide to use manual lookup for some capabilities and automatic updates for others. Additionally, the update dynamics of each capability can be changed individually from manual lookup to automatic updates.

In the following, we describe manual lookup and automatic updates of capabilities in detail, and how users can switch dynamically between those mechanisms for individual capabilities.

Manual Lookup

The user can manage the local network address cache in each capability manually. For this purpose, users can resolve a service name by calling `lookup_once`. By calling this method, the service name is resolved exactly once. LibRIPC will instruct its ZooKeeper client to read the corresponding znode without setting a watch. There will be no notification when the data contained in the znode changes. Thus, the resulting capability will not be updated when the service is migrated or shut down. Sending messages using that capability works as long as the receiver

can be reached under the address that was returned by ZooKeeper.

Once the service is migrated to a different host, the user has to query the current network address manually. There are two ways to obtain up-to-date information about a service. The first is to simply resolve that service name again by calling `lookup_once` and using the newly created capability. Since almost all functions of our API expect a capability as parameter, we decided to design another function. To update an existing capability, users can pass it to `update_once`. This function uses the service name stored within the capability. Then, it reads the corresponding znode in the same way as `lookup_once` does. However, libRIPC then stores the data contained in that znode as network address in the supplied capability instead of creating a new one. Afterwards, the user-supplied capability can again be used to send messages until the receiver migrates.

Using manual lookup allows clients of a service to implement application-specific update policies at the cost of less abstraction from the network.

Automatic Updates

Handling failures in sending messages manually by repeatedly calling libRIPC's lookup or update functions adds management overhead to the application. Also, libRIPC currently uses unreliable messaging. To detect a failure in sending a message, users have to create their own timeout handling, which further increases implementation overhead.

Generally, users of libRIPC are not interested in changes in the physical location of a service in itself. They are interested in being able to communicate with it, regardless of which process on which host is currently providing it legitimately. Especially in scenarios where services are migrating often, processes might want to let libRIPC handle these situations.

For this reason, our API includes the `lookup` function. Similarly to `lookup_once`, this function will perform a resolution of a service name and return a capability. The difference, however, is that `lookup` sets a watch when reading the znode. When a process changes the network address of that service, it writes the new address into that znode. The ZooKeeper servers notify all ZooKeeper clients that currently have a watch for the data of this znode. Upon receiving the notification, the individual ZooKeeper clients execute libRIPC's watch event handler. Thus, libRIPC is notified by ZooKeeper when the network address of that service is changed. Upon receiving the notification, libRIPC reads the znode and renews the watch. LibRIPC then updates the corresponding capability. Thus, processes can always sent message to services that have been looked up via `lookup`, regardless of whether the service has migrated.

Additionally to updating capabilities automatically, `lookup` gives users the possibility to specify a callback function that expects a capability as parameter.

When libRIPC receives a notification, it updates the affected capabilities. For each capability that was updated, libRIPC will then execute the corresponding callback function.

This feature allows users to implement their own event handling. For example, `lookup` might have returned a capability to a service that was not running. In this case, a client of that service might want to wait until the service has started. By providing the mechanism of a callback function, clients do not have to rely on busy waiting ("polling"). Instead, they can implement proper waiting by using semaphores themselves.

The user can obtain several capabilities for each service by calling `lookup` or `lookup_once` several times. Thus, the user may obtain both static and dynamic capabilities for a certain service. When receiving the notification that a certain znode has changed, libRIPC must not update static capabilities because the user expects them not to change. On the other hand, we have to update all dynamic capabilities. If the user obtained multiple dynamic capabilities of the same service by calling `lookup` with different callback functions, libRIPC has to determine the correct callback to execute for each capability.

To be able to update each dynamic capability and execute the corresponding user callback in libRIPC's watch event handler, we created a watcher context data structure. The watcher context contains pointers to the corresponding capability and user-set callback function. LibRIPC creates a watcher context for each call to `lookup` and passes it to ZooKeeper's read function. Since the ZooKeeper client library passes the watcher context as parameter to libRIPC's watch handler, libRIPC can easily determine which capability to update and which user-supplied callback to execute.

Since libRIPC reads the corresponding znode again on each notification it receives, automatic updates can create lookup overhead. If the user sends messages infrequently, libRIPC will request the new network address of the service once the services changes its address, without the user ever sending a message to that particular address.

Managing Update Dynamics

Each capability can either be updated automatically by libRIPC or explicitly via manual calls to update functions by the user. To change the update dynamics of a capability between automatic and manual, our naming scheme provides the functions `update` and `update_disable`.

When a user passes a capability to `update`, libRIPC updates the capability and ensures that it will be updated when the corresponding service changes its network address. Regardless of the original update dynamics of the supplied capability, that capability will from then on be automatically updated until libRIPC

is instructed otherwise. Similarly to `lookup`, the user can specify a callback function to be executed after that capability has been updated.

LibRIPC updates a capability by reading the corresponding znode and, in case of `update`, creating an appropriate watcher context. When the ZooKeeper client receives a notification that a certain znode has changed, it will call the watcher event handler of libRIPC once for each call of ZooKeeper's read function that passed a watcher context. The user is allowed to perform manual updates at any time. In particular, the user is allowed to perform multiple updates of the same capability. In this case, when the watch is triggered for the corresponding znode, the ZooKeeper client library calls libRIPC's watch event handler several times for a single capability even though only one watch was triggered. Without precaution, libRIPC would execute the user-supplied callback several times for that capability. To ensure that libRIPC executes that callback only once for each capability, we added a boolean flag to the watcher context data structure. This flag determines whether that individual watcher context is still enabled. When the user calls `update` on a capability that already has a watcher associated, libRIPC creates a new watcher context and disables the old one. On watcher events, libRIPC will only update the capability referenced by the supplied watcher context if the watcher context is enabled.

Similarly, users can pass capabilities that are currently to be updated automatically to `update_disable`. LibRIPC simply disables the watcher context and returns control to the user. When the ZooKeeper server sends a notification to the ZooKeeper client, the internal event handler of libRIPC will still be called with that watcher context. LibRIPC recognizes that the watch is disabled and will not read the znode corresponding to the capability referenced by the watcher context. Thus, libRIPC will not set another watch and no further notification will be generated for this capability.

ZooKeeper watches are only triggered once. Provided that a certain watcher context was passed to ZooKeeper's read function only once, once ZooKeeper has called libRIPC's event handler with that watcher context as parameter, it will not use that watcher context again. When the event handler of libRIPC is called with a watcher context that has been disabled, libRIPC will not perform another read of that znode and, thus, not renew the watch. Instead, libRIPC frees the memory allocated for the watcher context.

Thus, after `update_disable` returns, the capability passed by the user will not be updated if the corresponding service changes its network address, even if that change triggers an existing ZooKeeper watch.

4.5 Future Work

After receiving a notification, libRIPC is aware that the locally stored address of the service is outdated. To prevent silent failures when the user tries to send a message, libRIPC could internally queue all messages the user tries to sent during this time via that capability. When libRIPC has retrieved the new address of that service, it could send the messages contained in that queue.

During migration, a service will be offline temporarily. When a service goes offline, it clears the address stored in the corresponding znode. LibRIPC will receive a notification and read the corresponding znode. After reading the znode, libRIPC is aware that the service is offline. By having set a watch, libRIPC will receive another notification once the address of that service changes again. Thus, libRIPC could continue queueing messages sent via that capability until the service is online again.

Chapter 5

Evaluation

To evaluate the performance of our prototype, we implemented a simple client-server application. On one node of our cluster, we started a process providing a message acknowledgement service. This service listens for messages and responds to each one with an acknowledgement message containing the string `ACK`. We started client processes on other nodes of this cluster, at most one client per node. These processes resolve the network address of the service and send messages containing a four byte wide integer to the server. After each message a client has sent, it waits for an acknowledgement before sending the next message.

We evaluated the performance of both the old broadcast-based resolver and our prototype, both with and without using resolver caching.

We performed each of these experiments using two, three and four nodes of the cluster. In each case, we started all clients at the same time. All evaluations of our prototype were done using a ZooKeeper ensemble consisting of one ZooKeeper server per node.

5.1 Short Message Send Performance

We compared the current implementation of libRIPC, using its broadcast-based resolver and cache, against our prototype, re-using the capability that the client has looked up. In each run, the client sent 1000000 messages to the service and received the same amount of acknowledgement messages.

Relying on the local resolver cache or re-using a capability obtained by one of the lookup functions is the default usage pattern of libRIPC. After the client has sent its first message, the network address is contained in the resolver cache. Respectively, after initial lookup, the network address is contained in the capability. Due to the fact that resolution is performed only when sending the first message, we expect performance of current libRIPC and our prototype to be equal. We

expect runtime to depend almost exclusively on time required to send and acknowledge messages.

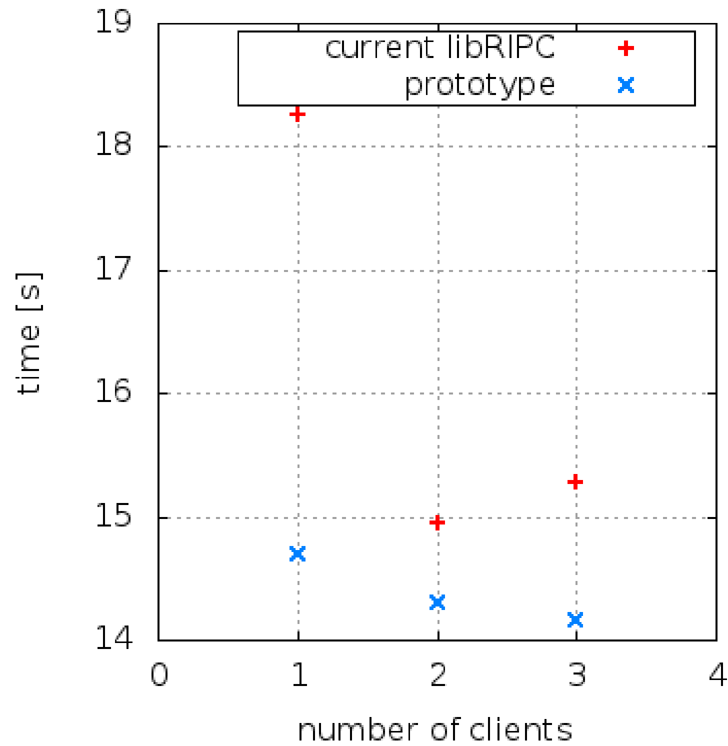

Figure 5.1: Time required to send and acknowledge 1000000 messages

However, Figure 5.1 shows that our prototype was even faster than the current implementation of libRIPC. One reason for this result is that we started measuring time required to send all messages after initialization of the client. Our prototype has, at that point, already performed its resolution. Since resolution of service IDs is done implicitly, measurement of time required to send these messages includes time required to perform one broadcast-based resolution. However, a single broadcast should not take this much time. Thus, further reasons for this result remain to be investigated.

Nevertheless, we conclude that using capabilities as token to address services does not add overhead to message sending.

5.2 Lookup Performance

We evaluated our prototype with individual lookup before each message against the broadcast-based resolver without using its cache. We extended the previous experiment to perform a resolution of the service prior each message sent from the client. In each run of the experiment, we sent 10000 messages from both client and server and performed the 10000 lookups of the service by the client. Comparing these results against the results we obtained by using caches and re-using capabilities, we can determine the amount of lookup overhead caused by resolution.

To disable the resolver cache of libRIPC on the client-side, we cleared the client's cache entry for the service ID of the acknowledgement service after each acknowledgement message received. To evaluate the resolution mechanism of our prototype, the client calls `service_update_once` before each message it sends.

We use `service_update_once` to prevent overhead caused by allocating new capability data structures in `service_lookup_once`. Since the service will not migrate in our case, the client is not interested in having libRIPC automatically keep its capability up-to-date. Thus, we use `service_update_once` instead of `service_update` to prevent allocation of watcher contexts that will not be used.

Figure 5.2 shows that, with increasing number of clients simultaneously performing lookups using our prototype, time required to perform those lookups increases exponentially. Also, resolving a service name by using our prototype takes about 35 times as long as resolving a service ID by using libRIPC's current resolver, even though we used one ZooKeeper server on each node.

There are several reasons for these results. First, we did not implement the prototype with performance in mind. For example, every single time libRIPC reads a znode it generates the path name to be passed to ZooKeeper again. LibRIPC allocates a buffer for the path, writes a forward slash into it, and appends the service name that is stored in the capability. For optimization, we could cache the path name in the capability The service name could then be obtained by reading the path name starting at its second character. Secondly, all communication required for resolution is done by ZooKeeper. ZooKeeper client and server communicate using ethernet and TCP/IP. This communication involves hardware of comparatively low performance, protocol overhead, and context switches. The broadcast-based resolver of libRIPC is handcrafted for service ID resolution and uses the infiniband hardware directly. Thirdly, the ZooKeeper client connects to an arbitrary server of the ensemble. As long as libRIPC performs only reads of znodes, network traffic could probably be decreased if each ZooKeeper client connects to the server instance on the local host. While these arguments could explain constant overhead, we have yet to identify the "bottleneck" which causes exponential growth.

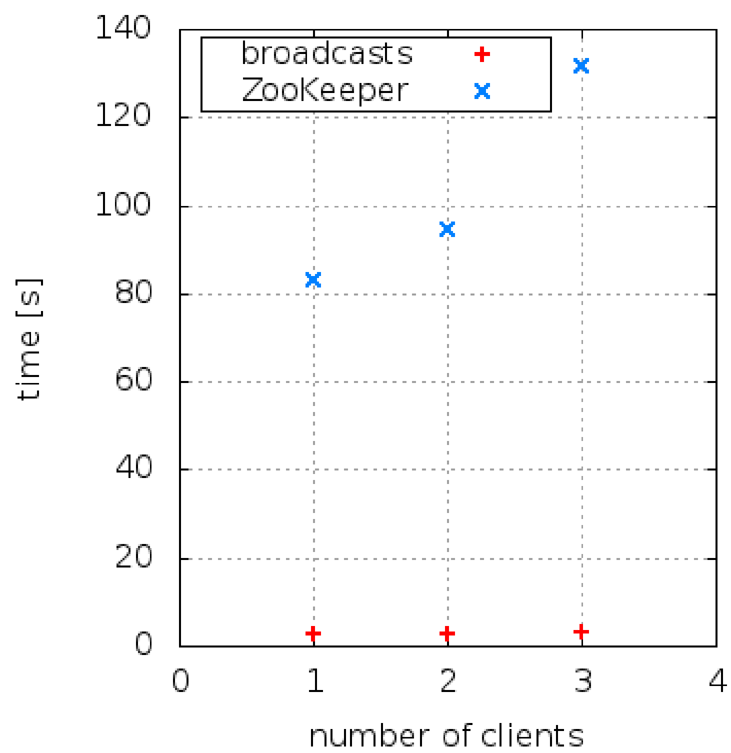

Figure 5.2: Time required to send and acknowledge 10000 messages and perform a resolution before each one

Chapter 6

Conclusion

In this thesis, we presented a naming scheme that allows users of libRIPC to identify services by descriptive names. We designed resolution of those name to be secure and scalable by using a distributed, central infrastructure in the network. To avoid overhead, resolution returns capabilities which are used and re-used by processes to address communication partners. Our naming scheme avoids message loss due to migration by notifying affected libRIPC instances when a service has changed its physical location. We implemented a prototype of this naming scheme using Apache ZooKeeper.

Evaluation showed that our prototype takes about 35 times the time to resolve a service compared to libRIPC's current broadcast-based resolver. However, addressing processes via capabilities instead of service IDs proved to add no overhead.

We therefore conclude that, while there remains work to be done, our naming scheme improves libRIPC to be better suited to fulfilling requirements of process addressing in future cloud applications.

Bibliography

[1] https://issues.apache.org/jira/browse/ZOOKEEPER-442. ZooKeeper feature request to remove watches.

[2] Jonathan Appavoo, Volkmar Uhlig, Jan Stoess, Amos Waterland, Bryan Rosenburg, Robert Wisniewski, Dilma Da Silva, Eric Van Hensbergen, and Udo Steinberg. Providing a cloud network infrastructure on a supercomputer. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 385–394, Chicago, Illinois, USA, June 21 2010.

[3] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *In USENIX Annual Technical Conference*.

[4] Jens Kehne, Marius Hillenbrand, Jan Stoess, and Frank Bellosa. Light-weight remote communication for high-performance cloud networks. In *Proceedings of the 1st IEEE International Conference on Cloud Networking*, Paris, France, November 28–30 2012.

[5] Scott Leberknight. Distributed Coordination With ZooKeeper Part 4: Architecture from 30,000 Feet. July 2013. http://www.nofluffjuststuff.com/blog/scott_leberknight/2013/07/distributed_coordination_with_zookeeper_part_4_architecture_from_30_000_feet.

[6] The Apache Software Foundation. ZooKeeper Programmer's Guide: Developing Distributed Applications that use ZooKeeper. http://zookeeper.apache.org/doc/r3.4.5/zookeeperProgrammers.html. Official documentation bundled with ZooKeeper version 3.4.5.

[7] Zouheir Trabelsi and Wassim El-Hajj. Arp spoofing: a comparative study for education purposes. In *2009 Information Security Curriculum Development Conference*, InfoSecCD '09, pages 60–66, New York, NY, USA, 2009. ACM. http://doi.acm.org/10.1145/1940976.1940989.

[8] ZooKeeper. zookeeper/zookeeper.jute.h. Internal header file of the ZooKeeper client C binding (version 3.4.5).