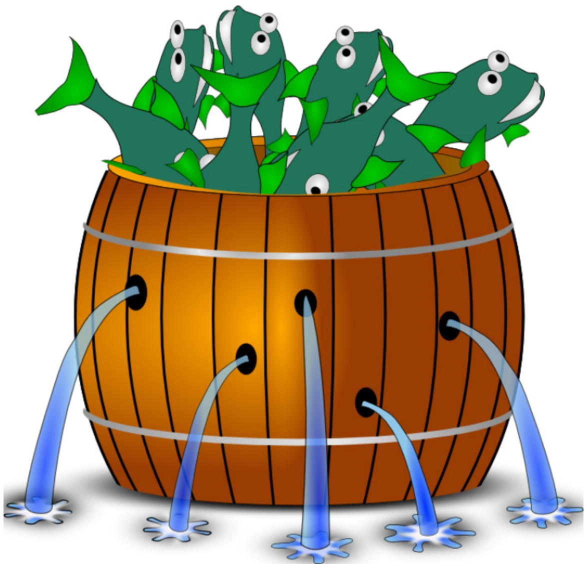
Barrelfish Project
ETH Zurich



Capability Management in Barrelfish

Barrelfish Technical Note 013

Akhilesh Singhania, Ihor Kuz

N/A

Systems Group, ETH Zurich
c/o Department of Computer Science
ETH Zurich CAB F.79, Universitätstrasse 6, Zurich 8092, Switzerland
`http://www.barrelfish.org/`

Revision History

Revision	Date	Author(s)	Description
1.0	08.03.2011	AS	Initial version

Chapter 1

Introduction

This document discusses the state of capabilities in the Barrelfish operating system.

Chapter **??** lists the currently known issues with capability management and **??** discusses the type system.

Chapter **??** discusses the current state of the implementation in Barrelfish, chapter **??** discusses different approaches for maintaining a multicore mapping database of capabilities, chapter **??** discusses the requirements from a correct solution and discusses four different solutions, chapter **??** discusses some Barrelfish specific challenges in implementing the solutions, and chapter **??** highlights issues not yet covered in this document.

Chapter 2

Known Issues

- Kernel operations should not take longer than $O(1)$. Some capability operations are not constant time but can take $O(n)$ where n is the size of the mapping database. Spending so much time in the kernel is detrimental for good scheduling as when in the kernel, interrupts are disabled. All non constant time operations should be punted to the monitor.
- When the last copy of a lmp capability or frame capability backing ump channels is deleted, should it initiate a connection tear down?
- We have no mechanics for tracking frames mapped into VNodes. So that if either is deleted, the entries are properly cleaned up.
- Multicore capability management. We have an implementation on the tip. It is not enabled and has some shortcomings. Not all shortcomings are not known however some missing details are known. As discussed in chapter **??**, all remote relations including descendants, copies, and ancestors must be tracked. The implementation only tracks ancestors.
- Fragmentation of memory. Example: a ram capability of 4GB is retyped into two capabilities of 2GB each. One of the 2GB capability is deleted. The only way to get a capability to that 2GB back is to also delete the other 2GB capability and then retype the 4GB capability again.
- The mapping database is maintained as a doubly linked list. Implementing Boolean operations like "has_descendants" and "has_copies" require walking the database and testing type specific data structures. We feel that these are not efficient or robust. If we move the mapping database into the monitor, we think that this should at worse become a performance issue and not a correctness one.
- IO space can be treated in a same way as physical memory. So instead of using mint operations to update ranges, we use retype operations. This is useful as different IO capabilities will have the ancestor, descendant, and copy relationships rather than just a copy relationship. The former is richer and can offer improved maintenance.
- When a new capability is introduced in a core via cross-core send, the kernel must walk the entire mapping database to find the appropriate place to insert the capability. The operation is linear time. This operation must either happen in the monitor or must be made more efficient. If we move the mapping database into the monitor, we think that this should at worse become a performance issue and not a correctness one.
- Simon and I had envisioned a memory server mechanism where it would delete all ram capabilities after handing them out. This allows Simon to implement his version of domain shutdown (discussed in a supporting document). In a nutshell, in his system, the spawn daemon saves all user dispatcher capabilities and revokes them to kill a user dispatcher and repossess its memory. However, there is a subtle problem with this mechanism. A malicious dispatcher can create loops in its CSpace. E.g. root cnode contains a capability to cnode1, cnode1 contains a capability to cnode2, and cnode2 contains another capability to cnode1. When a dispatcher does this and the OS tries to kill it, root cnode will be deleted and all capabilities within it as

well. When the OS deletes cnode1, it will check if it has copies, which it does so cnode1 will not be deleted. This will leak memory in the system.

To avoid this problem, the OS must maintain ancestors of all capabilities that are handed out to user applications and to kill the user application, it must revoke the ancestor capabilities.

Chapter 3

Type system

In this chapter, we cover the type model of capabilities and the supported types in Barrelfish.

3.1 Type Model

[*We do not implement capability rights yet.*]

Name Each type has a unique name.

Origin A capability type is either *primitive*, which means that capabilities of the type may be created only through a special process (eg. at boot time), or *derived*, which means that capabilities of the type may be created by retyping an existing capability. For primitive types, we specify how the capabilities of that type are created; for derived types, we specify which capability types may be retyped to yield a capability of the given type.

Retypability Some types of capability may be *retyped* to create one or more new capabilities of the same or different type. If this is the case, we specify for each type from what other types of capability it may be retyped.

Mint parameters It is possible to specify type-specific parameters when *minting* capabilities. We specify for each type the interpretation of the type-specific parameters. When not specified, they are ignored.

Interpretation of rights The interpretation of the primitive capability rights is type-specific. A capability type defines the interpretation of these rights, usually in the specification of the legal invocations.

Transferability to another core Depending on its type, it may or may not be possible to transfer a capability to another core.

Last copy deleted The type specific operations to perform when the last copy of a capability is deleted. For capability types that refer to actual memory, if the last reference to a piece of memory is deleted, then the memory must be garbage collected.

Concrete representations Each capability type has one or more representations: in the memory of each core on which it may appear, and in a canonical serialised representation used for transferring it in a message between cores. These are specified as Hamlet[**?**] data types.

Invocations Most capability types support one or more type-specific operations through the use of the invoke system call. (documented in **??**).

3.2 Types

3.2.1 CNode

A CNode refers to an array of capabilities of some power-of-two size. CNodes are used to hierarchically construct the CSpace of a domain, as described in **??**. All capability management is performed through invocations on CNodes.

A CNode capability stores a *guard* and *guard size*, which is expressed as a number of bits. As in guarded page tables[**?**], the guard allows the depth of the CSpace tree to be reduced by skipping levels that would only contain a single mapping. When resolving a CSpace address that has led to a CNode capability, the guard is compared with the corresponding number of bits from the remaining part of the address, and if it does not match, the lookup fails.

Many CNode invocations require that the caller provide both a CSpace address and the number of *valid bits* in the address. This allows the invocations to refer to another CNode capability that is located at an intermediate level in the tree, and thus would usually be recursed through by the address resolution algorithm. If the number of valid bits associated with a CSpace is less than the size of a full CSpace address, only the least significant bits that are valid, but starting with the most significant bit thereof, are used, and the address lookup terminates early.

Origin Retyping from RAM type capabilities

Retypability No

Mint parameters The mint parameters can be used to set a guard on a CNode.

- Parameter 1: The guard to set.
- Parameter 2: The size of the guard in bits.

Interpretation of rights [*Explain rights and rights mask. Capability rights and rights masks are currently not implemented. This means that every user domain holding a capability has full rights to it.*]

Transferability to another core Yes. When transfered to another core, capability is implicitly retyped to a Foreign CNode type. [*We do not allow CNode type caps to be transferred yet.*]

Last copy deleted When the last copy is deleted, all capabilities stored within it are also deleted.

Concrete representations The in-memory representation of on x86-64 is as follows:

```
datatype cnode_cap "CNode capability" {
  cnode 64 "Physical base address of CNode";
  bits 8 "Number of bits this CNode resolves";
  rightsmask 8 "Capability rights mask";
  guard_size 8 "Number of bits in guard word";
  guard 32 "Bitmask already resolved when reaching this CNode";
};
```

Mint invocation

Argument 1: CSpace address of destination CNode

Argument 2: Slot number in destination CNode

Argument 3: CSpace address of source capability

Argument 4: Number of valid bits in destination CNode address

Argument 5: Number of valid bits in source capability address

Argument 6: Type-specific parameter 1

Argument 7: Type-specific parameter 2

Table 3.1: Permissible types for the Retype invocation

Source	Destination	Variable size?
Physical address range	Physical address range	Yes
Physical address range	RAM	Yes
Physical address range	Device frame	Yes
RAM	RAM	Yes
RAM	CNode	Yes
RAM	VNode	No
RAM	Dispatcher	No
RAM	Frame	Yes
Dispatcher	IDC endpoint	No

The Mint invocation creates a new capability in an existing CNode slot, given an existing capability. The new capability will be a copy of the existing capability, except for changes to the *type-specific parameters*.

The use of the two type-specific parameters is described along with the description of the relevant type.

If the destination capability is of type VNode, then instead of a mint, a page table entry is made into the page table pointed to by the VNode. In this case, the source capabilities must be of type VNode, Frame, or Device Frame. The first parameter specifies architecture specific page (table) attributes and the second parameter specifies an offset from the source frame/device frame to map.

Copy invocation

Argument 1: CSpace address of destination CNode

Argument 2: Slot number in destination CNode

Argument 3: CSpace address of source capability

Argument 4: Number of valid bits in destination CNode address

Argument 5: Number of valid bits in source capability address

This invocation is similar to Mint, but does not change any type-specific data.

If the destination capability is of type VNode, then instead of a copy, a page table entry is made into the page table pointed to by the VNode. In this case, the source capabilities must be of type VNode, Frame, or Device Frame.

Retype invocation

Argument 1: CSpace address of source capability to retype

Argument 2: Type of new objects to create

Argument 3: Size in bits of each object, for variable-sized objects

Argument 4: CSpace address of destination CNode

Argument 5: Slot number in desination CNode of the first created capability

Argument 6: Number of valid bits in destination CNode address

This invocation creates one or more new descendant capabilities of the specified type in the specified slots, given a source capability and a destination type. It will fail if the source or destination are invalid, or if the capability already has descendants. (some capability types, currently only the dispatcher type can be retyped even if it already has descendants). The destination slots must all occupy the same CNode. The permissible source/destination pairs are shown in **??** and **??**. The number of new capabilities created is determined by the size of the source capability divided by the size of the newly-created objects.

Figure 3.1: Valid capability retyping paths

Delete invocation

Argument 1: CSpace address of capability to delete

Argument 2: Number of valid bits in capability address

This invocation deletes the capability at the given address, freeing the associated CNode slot.

Revoke invocation

Argument 1: CSpace address of capability to Revoke

Argument 2: Number of valid bits in capability address

This invocation revokes the capability at the given address.

The capability itself is left untouched while all its descendants and copies are deleted.

3.2.2 Foreign CNode

[*This has not been implemented yet*]

The foreign CNode capability gives a domain on a core the ability to specify a capability that actually resides on another core. This capability allows for the holder to create local copies of the capabilities stored in the actual CNode modulo rights as can be implemented. The capability tracks on which core the actual CNode resides. [*Full implementation and discussion pending*]

Origin When a CNode capability are copied to another core.

Retyability No

Mint parameters None

Interpretation of rights [*Explain rights*]

Transferability to another core Yes

Last copy deleted [*NYI*]

Concrete representations The in-memory representation on x86-64 is as follows:

```
datatype fcnode_cap "Foreign CNode capability" {
  cnode       64  "Physical base address of CNode";
  bits         8  "Number of bits this CNode resolves";
  rightsmask   8  "Capability rights mask";
  core_id      8  "Core id of the core the actual CNode capability
                   resides in";
  guard_size   8  "Number of bits in guard word";
  guard       32  "Bitmask already resolved when reaching this CNode";
};
```

[*Discussion pending on invocations*]

3.2.3 Physical address range

Most domains will generally not handle capabilities of this type. They are introduced because the kernel relies on the user-space software to decide the location of RAM.

By retyping physical address range capabilities to RAM, the caller guarantees that the underlying region does contain RAM that can safely be used for storage of kernel objects. Any domain with access to physical address range capabilities is therefore a critical part of the trusted computing base.

Origin Created at boot time in the bsp core based on the multiboot info.

Mint parameters None

Retyability To Physical address range, RAM or DevFrame type.

Interpretation of rights [*Explain rights*]

Transferability to another core Yes

Last copy deleted [*NYI, maybe inform some special dispatcher like memory server*]

Concrete representations The in-memory representation on x86-64 is as follows:

```
datatype physaddr_cap "Physical address range capability" {
  base       64  "Physical base address of region";
  bits        8  "Size of region";
};
```

3.2.4 RAM

A RAM capability refers to a naturally-aligned power-of-two-sized region of kernel-accessible memory.

Origin Retyping from physical address range capabilities.

Retyability To RAM, Frame, CNode, VNode, or Dispatcher types.

Mint parameters None

Interpretation of rights [*Explain rights*]

Transferability to another core Yes

Last copy deleted [*NYI, maybe inform some special dispatcher like memory server*]

Concrete representations The in-memory representation on x86-64 is as follows:

```
datatype ram_cap "RAM capability" {
  base       64  "Physical base address of region";
  bits        8  "Size fo region";
};
```

3.2.5 Dispatcher

This capability type refers to the kernel object associated with a user-level dispatcher (see **??**).

Origin Retyping from RAM capabilities.

Retyability To IDC Endpoint type

Mint parameters None

Interpretation of rights [*Explain rights*]

Transferability to another core No

Last copy deleted [*NYI, maybe inform some special dispatcher like spawn daemon*]

Concrete representations The in-memory representation on x86-64 is as follows:

```
datatype dcb_cap "Dispatcher capability" {
  dcb         64  "Pointer to the in kernel representation of
                   the dispatcher control block";
};
```

Setup invocation

Argument 1: CSpace address of domain's root CNode (root of CSpace)

Argument 2: Number of valid bits in root CNode address

Argument 3: CSpace address of domain's root VNode (root page table)

Argument 4: Number of valid bits in root VNode address

Argument 5: CSpace address of dispatcher frame (user-level dispatcher data) capability

Argument 6: Whether to make dispatcher runnable

This invocation sets any of the above parameters on a dispatcher object. If any of the CSpace addresses are null, they are ignored. Additionally, once all of the parameters are set (either in a single invocation, or after multiple invocations), and if the runnable flag is set, the dispatcher is made runnable. [*There are additional invocations in the code that we have not discussed yet.*]

3.2.6 IDC Endpoint

Every IDC endpoint refers both to a dispatcher and an *endpoint buffer* within that dispatcher. The endpoint buffer is specified as an offset from the start of the dispatcher frame, and is the location where the kernel delivers IDC messages. It is also delivered to the user with an LRPC message. The initial endpoint offset of an IDC endpoint capability when it is retyped from a dispatcher capability is zero; the capability cannot be used to send IDC until the the offset is specified changed by minting an endpoint to another endpoint.

Origin Retyping Dispatcher type capabilities.

Mint parameters The mint parameters can be used to change the badge on the capability

- Parameter 1: The endpoint offset to set on the capability.

Retyability No

Interpretation of rights [*Explain rights*]

Transferability to another core No

Last copy deleted [*NYI, inform some entity to initiate connection teardown*]

Concrete representations The in-memory representation on x86-64 is as follows:

```
datatype idc_cap "IDC endpoint capability" {
  listener      64  "Pointer to the in kernel representation of the
                     receiver's dispatcher control block";
  epoffset      64  "Offset of endpoint buffer within dispatcher
                     structure";
};
```

Invocation Any invocation of an endpoint capability causes the entire message to be delivered to the dispatcher to which the endpoint refers (see **??**).

3.2.7 VNode

A VNode capability refers to a hardware page table and is used to manage a domain's virtual address space. Frame and device frame capabilities can be copied or minted into them or deleted from them by invoking a CNode. The architecture may impose limitations on the capabilities that may be copied into a VNode, or may allow extra attributes to be set when minting; see **??**.

Origin Retyping from RAM type capabilities.

Retyability No

Mint parameters None

Interpretation of rights [*Explain rights*]

Transferability to another core [*Discussion pending*]

Last copy deleted [*NYI, initiate mechanisms to unmap from associated page tables and remove mapped in page tables and frames*]

Concrete representations The in-memory representation on x86-64 is as follows:

```
datatype vnode_cap "VNode capability" {
  base      64  "Base address of the page table";
};
```

3.2.8 Frame

A frame capability refers to a naturally-aligned power-of-two-sized region of physical memory that may be mapped into a domain's virtual address space (by copying it to a VNode). When a frame capability is created (ie. retyped from RAM), the kernel zero-fills the frame. [*Is this a good idea? Shouldn't we be able to pre-zero frames? -AB*]

Origin Retyping from RAM type capabilities.

Retyability To Frame type

Mint parameters None

Interpretation of rights [*Explain rights*]

Transferability to another core Yes

Last copy deleted [*NYI, initiate unmapping from page tables. We may choose for this to happen when the last copy of a frame within a dispatcher is deleted rather than the last copy in the entire system.*]

Concrete representations The in-memory representation on x86-64 is as follows:

```
datatype frame_cap "Frame capability" {
  base        64  "Physical base address of untyped region";
  bits         8  "Size of the region";
};
```

Identify invocation This invocation returns the physical address and size (in bits) of the frame.

3.2.9 Device frame

A device frame capability refers to a naturally-aligned power-of-two-sized region of physical address space that may be mapped into a domain's virtual address space (by copying it to a VNode). Unlike frame capabilties, the kernel does not zero-fill device frame capabilities upon mapping. As the name implies, device frames are typically used for access to memory-mapped devices.

Origin Retyping Physical address range type capabilities.

Retyability To Device frame type

Mint parameters None

Interpretation of rights [*Explain rights*]

Transferability to another core Yes

Last copy deleted [*NYI, initiate unmapping from page tables. We may choose for this to happen when the last copy of a frame within a dispatcher is deleted rather than the last copy in the entire system.*]

Concrete representations The in-memory representation on x86-64 is as follows:

```
datatype device_cap "Device Frame capability" {
  base        64  "Physical base address of region";
  bits         8  "Size of the region";
};
```

Identify invocation This invocation returns the physical address and size (in bits) of the frame.

3.2.10 IO

IO capability gives the holder the ability to read and write to IO ports.

Origin A single capability created at boot time in the bsp core.

Retyability No

Mint parameters Used to specify the region of io space the capability can access.

- Parameter 1: Start of the region
- Parameter 2: End of the region

Interpretation of rights [*Explain rights*]

Transferability to another core Yes

Last copy deleted [*NYI*]

Concrete representations The in-memory representation on x86-64 is as follows:

```
datatype io_cap "IO capability" {
  start       16  "Start of the granted IO range";
  end         16  "End of the granted IO range";
};
```

Outb invocation

Argument 1: IO port number

Argument 2: Output data

This invocation writes a byte to the the specified IO port

Outw invocation

Argument 1: IO port number

Argument 2: Output data

This invocation writes a two byte word to the the specified IO port

Outd invocation

Argument 1: IO port number

Argument 2: Output data

This invocation writes a four byte to the the specified IO port

Inb invocation

Argument 1: IO port number

This invocation returns a byte read from the specified IO port.

Inw invocation

Argument 1: IO port number

This invocation returns a 16-bit word read from the specified IO port.

Ind invocation

Argument 1: IO port number

This invocation returns a 32-bit doubleword read from the specified IO port.

3.2.11 IRQ table capability

The IRQ table capability allows the holder to configure the user-level handler dispatcher that will be invoked when the kernel receives device interrupts.

Origin Given to the first domain spawned on a core.

Retyability No

Mint parameters None

Interpretation of rights [*Explain rights*]

Transferability to another core No

Last copy deleted [*NYI*]

Concrete representations This capability type has no representation associated with it as it is used to simply give permissions to the holders and does not refer to any kernel data structure.

Set invocation

Argument 1: IRQ number

Argument 2: CSpace address of asynchronous endpoint capability

This invocation sets the user-level handler endpoint that will receive a message when the given interrupt occurs. While a handler is set, interrupts will be delivered as IDC messages, as described in **??**.

Delete invocation

Argument 1: IRQ number

This invocation clears the handler for the given IRQ.

3.2.12 Kernel Capability

So far, this capability is treated as the magic capability that gives the holder a backdoor into performing special operations in the kernel.

Origin Given to the first domain spawned on a core.

Retyability No

Mint parameters None

Interpretation of rights [*Explain rights*]

Transferability to another core No

Last copy deleted [*NYI*]

Concrete representations The in-memory representation on x86-64 is as follows:

```
datatype kernel_cap "Kernel capability" {
  kernel_id    8   "Id of the Kernel";
};
```

Spawn core invocation

Argument 1: Apic ID of the core to try booting

Argument 2: CSpace address of the RAM capability to use to relocate the new kernel

Argument 3: CSpace address of the Dispatcher capability of the first domain to run

Argument 4: Number of valid bits for the root CNode to associate with the Dispatcher capability

Argument 5: CSpace address of the root CNode to associate with the Dispatcher capability

Argument 6: CSpace address of the VNode to associate with the Dispatcher capability

Argument 7: CSpace address of the dispatcher frame to associate with the Dispatcher capability

The invocation requests the kernel to try booting another core. The kernel is to be relocated into the given memory region and to run the the given domain.

Get core ID invocation

Argument 1: None

The invocation returns the APIC ID of the core.

Identify capability invocation

Argument 1: CSpace address of the capability to identify

Argument 2: Number of valid bits in the capability

Argument 3: Location of buffer to hold capability representation

The invocation stores the kernel's in-memory representation of the capability into the given buffer.

Identify CNode, get capability invocation

Argument 1: In memory representation of a CNode capability

Argument 2: Slot number of a capability within the CNode capability

Argument 3: Location of buffer to hold capability representation

The invocation stores the kernel's in-memory representation of the capability located at the given slot in the given CNode into the given buffer.

Create capability invocation

Argument 1: In memory representation of a capability

Argument 2: CSpace address of the CNode the place the created capability in

Argument 3: Number of valid bits in the CSpace address of the CNode

Argument 4: Slot number to place the capability in

Creates the given capability in the given slot in the given CNode.

Create capability invocation

Argument 1: In memory representation of a capability

Argument 2: CSpace address of the CNode the place the created capability in

Argument 3: Number of valid bits in the CSpace address of the CNode

Argument 4: Slot number to place the capability in

Creates the given capability in the given slot in the given CNode.

[*The other invocations are outdated and will probably change when the monitors are discussed*]

Chapter 4

Current State

This chapter will cover how capabilities are stored and what happens on capability invocation.

4.1 Storage

For security reasons, capabilities are stored in kernel-space and users are given pointers to them. Each capability is stored in two separate databases:

- Each dispatcher has an associated capability space that holds all the capabilities it has. The capability space of a dispatcher is implemented using the CNode type capability. Each dispatcher is associated with a "root CNode" that contains all capabilities the dispatcher has.
- Each core has a mapping database that holds all the capabilities on the core. The mapping database is implemented using a tree of the capabilities. As discussed later in the chapter, the mapping database stores the capabilities in a particular order to facilitate different capability operations.

4.2 Capability invocation

When a dispatcher invokes a capability, it passes the kernel an address of the capability in the CSpace it wishes to invoke. The kernel locates the capability starting from the dispatcher's root CNode (walks the capability space), verifies that the requested operation can indeed be performed on the specified capability and then performs the operation.

4.3 Data structures

Capabilities in Barrelfish are represented by the following data structures:

```
    struct mdbnode {
      struct cte *left, *right;  // Links to the mapping database
  ...
    };

    struct CNode {
      paddr_t cnode;      // Base address of CNode
      uint8_t bits;       // Size in number of bits
      ...
    };

    union capability_u {  // Union of all types of capabilities
      ...
      struct CNode cnode;
      ...
```

```
};

struct capability {
  enum objtype type;     // Type of capability
  union capability_u u; // Union of the capability
};

struct cte {
  struct capability   cap;     ///< The actual capability
  struct mdbnode      mdbnode; ///< MDB node for the cap
};
```

A capability, `cte`, consists of the actual capability represented by the "capability" structure and an entry in the mapping database represented by the "mdbnode" structure. The capability structure contains the type specific information and the mdbnode contains pointers for the tree representing the mapping database.

Capabilities can be looked-up in two ways.

- All capabilities on a core are stored in a mapping database. It is possible to reach any capability on the core by traversing from any other capability on the core.
- Capabilities are also stored in the CNode type capability. The area of memory identified by the CNode structure is actually an array of capabilities. Starting from the "root CNode" of a dispatcher, it is only possible to reach any capability the dispatcher holds.

4.4 Terminology

This section discusses some terminology to facilitate the discussion of capability management.

4.4.1 Copy

A capability X is a copy of a capability Y if:

- X was copied from Y
- or Y was copied from X
- or X was copied from Z and Z was copied from Y

4.4.2 Descendants

A capability X is a descendant of a capability Y if:

- X was retyped from Y
- or X is a descendant of Y1 and Y1 is a copy of Y
- or X is a descendant of Z and Z is a descendant of Y
- or X is a copy of X1 and X1 is a descendant of Y

4.4.3 Ancestor

A is a ancestor of B if B is a descendant of A.

4.5 CNode invocations

Most Capabilities have type specific invocations. Operations on the CNode capability modifies the capability space of the system. We discuss how these operations are implemented for a single core system here.

[*Invocations on other capability types will probably also modify the capability space but alas we don't know how those will work yet.*]

4.5.1 Retype

Retyping a capability creates one or more descendants of the capability. This operation will fail if the capability already has descendants. The descendants are inserted into a CNode specified by the operation and into the mapping database right after the retyped capability.

When a dispatcher issues the retype invocation, the kernel must traverse the mapping database to ensure that the capability has no descendants, create the descendants capabilities, insert them in the specified CNode and in the mapping database.

4.5.2 Copy

Copying a capability creates a new copy of it. The kernel walks the capability space to find the capability to be copied, creates the copy, and inserts it into the specified CNode and mapping database.

4.5.3 Delete

Delete removes the specified capability from the CNode in which it resides and from the mapping database. This operation cannot fail.

The kernel first walks the capability space to locate the capability to delete. It then walks the mapping database to check if there still exist copies of the deleted capability. If no copies are found, then it performs certain operations based on the capability type.

4.5.4 Revoke

Revoking a capability calls delete on all copies and descendants of it. When the operation returns, the capability will not have any copies or descendants.

The kernel walks the capability space to find the specified capability, uses the mapping database to find all copies and descendants of the specified capability and deletes them.

4.5.5 Looking up local copies and descendants

Due to the capability ordering used by the mapping database, copies are located adjacant to a capabality and descendants immediately thereafter. Therefore, it is easy to look up all related copies of a capability on the same core. This facilitates revocation by looking up all copies and descendants, retypes by checking for existing descendants, and deltes by checking for copies.

The following pseudo-code looks up all descendants and copies of a capability given the existence of type specific is_copy and is_descendant functions.

```
// Traverse forward
mdbnode *walk = successor(cap);
while (walk) {
  // Check if descendant
  if (is_descendant(cap, walk)) {
      // Found a descendant
      goto increment;
    }

    // Check if copy
    if (is_copy(cap, walk)) {
      // Found a copy
      goto increment;
    }

    // Cap is not a descendant or copy
    break;

  increment:
    walk = successor(walk);
  }

  // Traverse backwards
  mdbnode *walk = predecessor(cap);
  while (walk) {
    // Predecessors cannot be descendants

    // Check if copy
    if (is_copy(cap, walk)) {
      // Found a copy
      goto increment;
    }

    // Cap is not a copy
    break;

  increment:
    walk = predecessor(walk);
  }
}
```

4.6 Multicore extensions

The model above only works for a single core system. We have already extended it to work on multiple cores. Here we discuss these extensions.

4.6.1 Cross-core transfer

This is a special operation available only to the monitors. This sends a capability from one core to another.

4.6.2 Missing operations

When a capability is sent to another core, no state is maintained about it. When checking for capability relations in capability operations, only the current core is checked. Therefore these operations are not implemented correctly right now.

4.7 Summary

In this chapter, we presented a background and the current state of capabilities management in Barrelfish. We can now discuss different designs for multicore capability management.

Chapter 5

Maintaining the database

We consider the following approaches for managing the mapping database.

[*Use diagrams to better illustrate the discussion below.*]

- **No partition:** The mapping database and capability space for all cores is maintained on a single centralized coordinator. Accessing either the capability space or the mapping database on any core requires communication with the coordinator.
- **Partitioned data structure:** The mapping database for all cores is maintained on a single centralized coordinator and the capability space is maintained on the local cores. This implies that the cte structure is split between the coordinator and the local cores. Cores can access the capability space locally and have to message the coordinator to access the mapping database. Note that split "capability" and "mdbnode" structures need to maintain references to each other.
- **Centrally replicated space:** The mapping database and the capability space is replicated between a single coordinator and the local cores. This implies that two copies of each "cte" structure exist in the system, one on the local core and one on the coordinator. Both local core and the coordinator can access the mapping database and the capability space.
- **Partitioned space:** The mapping database and the capability space are maintained on the local cores. The entire "cte" structure can be accessed locally but capability operations will require coordination and communication with remote cores.
- **Minimal replication:** The database is partitioned between local cores as in the above approach and additionally for each capability the cores maintain a cache of its remote relations. This is discussed in more detail in section **??**.

We qualitatively compare the above approaches and why the partitioned space and minimal replication is the best.

5.1 Comparison

We compare the above approaches in this section.

5.1.1 Efficiency of capability invocations

When a dispatcher invokes a capability, the kernel has to look it up in the capability space starting from the dispatcher's "root" CNode. If the capability space is not local, then the core has to message the coordinator to perform the look up. A round trip with the coordinator is more expensive than a local look up and can become a scalability bottleneck if we use a single coordinator.

	Cap invocation	Local operations
No partition	-	-
Partitioned data structure	+	-
Centrally replicated space	+	-
Partitioned space	+	+
Minimal replication	+	+

Table 5.1: Summary of the different approaches.

The no partition approach does not maintain a local capability space and therefore may suffer from poor performance. All other approaches maintain the capability space locally.

5.1.2 Local operations

Approaches that enable more pure local operations will perform better as they will reduce the amount of cross-core communication. In the no partition, partitioned data structure, and replicated space approaches, no operation can be performed locally. In the partitioned and minimal replication approaches, certain operations such as copying capabilities is purely local.

5.1.3 Discussion

Table **??** summarizes our comparison of the five approaches. A (+) indicates that the approach performs relatively well on the given metric and a (-) indicates that the approach performs relatively poorly. Based on the results, we choose to implement the partitioned space and minimal replication approaches.

5.2 Caching

The minimal replication approach is characterized as the partitioned approach with caching the state of remote relations. Capabilities without remote relations are marked as such and when performing operations on these no remote communication is required.

When tracking remote relations, three types of relations must be tracked: copies, descendants, and ancestors. Tracking of remote copies and descendants is required so that revoke, retype, and delete operations can be correctly implemented. And capabilities must track their remote ancestors so if they are deleted, the remote ancestors can be informed to update the state about their remote descendants.

5.2.1 How to maintain the cache?

[*Once I discuss total order broadcast with caching, this discussion will be revamped.*]

There are two ways of maintaining the cache:

- **Single bit:** A bit each for the three types of remote relations is used. The bits merely indicate the presence of remote relations but provide no further information such as which cores have the remote relations.
- **List:** A list each for the three types of remote relations is used. The list contains exact information about which cores the remote relations exist on. Uhlig et al [?]'s core mask technique can be used to maintain the lists with fixed space requirement.

Comparison

Informing existing relations: When a capability that already has remote relations is sent to another core, with the single-bit approach, none of the existing cores with relations have to be informed, but with the list approach, cores with existing relations must be informed so they can update their lists.

With both approaches when the last local copy of a capability is deleted, its remote relations should be informed. This is required in the list approach so that the cores can update their list and is required in the single-bit approach so that the cores can determine if the last remote relation has been deleted and it can mark the capability as not having the remote relation anymore.

Number of cores communicated: With the single bit approach, all cores in the system must be contacted, while with the list approach, just the cores of interest must be contacted.

Discussion: The single-bit approach will probably be more efficient if the system has few cores and the capabilities that have remote relations, tend to have them on many cores. The list approach will have better performance if there are lots of cores in the system and typically a capability only has relations on a few cores.

5.2.2 Where to maintain the cache?

Imagine that there exists a capability with only local relations. Then cross-core transfer it applied to it. Should we only mark that capability as having a remote copy or should we mark all impacted capabilities accordingly? This section will discuss and compare these two approaches.

Mark all

When cross-core transfer is applied to a capability, it is marked has having remote copies and all its local relations are also marked as having the appropriate remote relations. Its local copies will be marked as having remote copies, its local descendants will be marked as having remote ancestors, and its local ancestors will be marked as having remote descendants.

All capabilities maintain complete state of their remote relations at all times.

Mark just individual capability

When cross-core transfer is applied to a capability, it is marked has having remote copies and none of its local relations are updated.

Capabilities do not maintain the complete state of their remote relations. Their local relations must be looked up to build the complete state of their remote relations.

Comparison

Both approaches have their specific strengths and weaknesses discussed here.

Cost of cross-core transferring: When applying cross-core transfer to a capability, in the mark all approach, all local relations must be updated whereas in the mark just individual capability approach, the local relations do not have to be updated. However, the cross-core transfer operation needs to send full information about the state of local relations so that the newly created remote capability can setup its cache properly. Gathering this information will require accessing all local relations so the mark all approach represents a small in the operation that must be performed anyways.

Cost of building state of remote relations: Any capability operation that requires information on the current state of remote relations will have to build the full state of remote relations for the capability. This is done trivially in the mark all approach as each capability maintains the full state. In the mark just individual capability approach, all local relations must be accessed to build the state.

Discussion: Given that the state of remote relations is trivially constructed in the mark all approach and the cost of cross-core transfer is only marginally higher than the bare minimum cost already required, we conclude that the mark all approach is superior to the mark just individual capability approach.

5.2.3 Summary

In summary, we have come up with two approaches for the type of cache: single-bit and list, and we have concluded that the mark all approach is superior for maintaining the cache.

Maintaining a cache will require the following adjustment to the capability operations presented above.

- When cross-core transfer is applied to a capability, it is marked as having remote copies. The operation sends information on the state of the capability's local and remote relations.
- When a capability is created due to the cross-core receive operation, it incorporates the information about relations sent with the cross-core transfer operation.
- When a copy of a capability is marked as having remote relations, the capability is marked as having the same remote relations.
- When a descendant of a capability is marked as having remote relations, the capability is marked also marked as having remote relations based on the following rules:
  - If the descendant has a remote copy, then capability has a remote descendant.
  - If the descendant has a remote descendant, then capability has a remote descendant.
  - If the descendant has a remote ancestor, then capability either has a remote copy or an ancestor depending on the outcome from the type-specific is_descendant function.
- When an ancestor of a capability is marked as having remote relations, the capability is marked also marked as having remote relations based on the following rules:
  - If the ancestor has a remote copy, then capability has a remote ancestors.
  - If the ancestor has a remote descendant, then capability either has a remote copy or a remote descendant depending on the outcome from the type-specific is_descendant function.
  - If the ancestor has a remote ancestor, then capability has remote ancestors.
- When a capability is retyped:
  - Its remote copies are marked as having remote descendants
  - The descendants are marked as having remote ancestors if the capability has remote copies.
- When a capability is copied, the copy is marked as having the same remote relations that the capability has.
- When the last copy of a capability on the core is deleted, its remote copies, descendants, ancestors are informed.

We now discuss some implications of maintaining a cache.

Drawbacks: Caching introduces the following drawbacks that the non-caching approach does not suffer from.

- The application dispatchers must first try to perform the capability operations locally and if that fails, then communicate with the monitors. If no caching were used, the application dispatchers can always directly communicate with the monitor saving one superfluous trip into the kernel.
- As discussed above, caching requires additional overhead in keeping the cache consistent.
- Caching increases the space requirement for maintaining the mapping database.

Caching can work if typically few capabilities are shared between cores and can hurt if many capabilities are shared.

Decide dynamically: It is clear that application scenarios will actually dictate if caching can help and if it does, which type of caching helps. To support this, Barrelfish can allow applications to specify which type of caching is well suited for it or the OS can gather appropriate metrics on application execution and based on that dynamically update the type of caching used. For this, we need to identify the cross-over point where one approach is preferred over the other.

Chapter 6

Solutions

In this chapter, we discuss mechanisms for implementing the capability operations. In section **??**, we will first discuss the challenges in correctly implementing the operations by discussing how executing multiple related operations in parallel can lead to conflicts and by discussing how an intuitive solution of using acknowledgments fails to resolve the conflicts. We then present the requirements from a correct solution in section **??** and compare four different correct solution in section **??**.

6.1 Challenges

We first discuss the conflicts that can arise if two or more related operations execute in parallel and then discuss an intuitive but incorrect solution of using acknowledgments to resolve the conflicts.

6.1.1 Conflicts

Performing overlapping operations on related capability can lead to conflicts. This section discusses the different types of conflicts that can arise.

Conflicts between two retypes, deletes, or revokes: If two different cores are trying to retype, delete, or revoke related capabilities, then they can conflict. If two different cores try to retype two copies of a capability, the correct behavior is for one to succeed and for one to fail. If two cores try to delete the last two copies of a capability, the correct behavior is for one to just delete locally and for the other to perform type-specific operations required when deleting the last copy of a capability. If two cores try to revoke the same capability, one should succeed, the other should fail and the capability it was trying to revoke should also be deleted.

Conflict between revoke and cross-core transfer: If a core is trying to revoke a capability, a copy or descendant of which another core is trying to transfer, the revoke operation should only finish when the in-flight capability is also deleted.

Conflict between revoke and retype: If a core is trying to revoke a capability, a copy or descendant of which another core is trying to retype, then the retype should either succeed and the new capabilities be deleted before revoke finishes or the retype should fail.

6.1.2 Non-conflicts

This section discusses why it is safe to perform other operations in parallel.

Delete and revoke: Delete at least deletes one capability and potentially more. If a revoke for a related capability is issued at the same time, they will overlap and potentially try to perform redundant operations but these are safe.

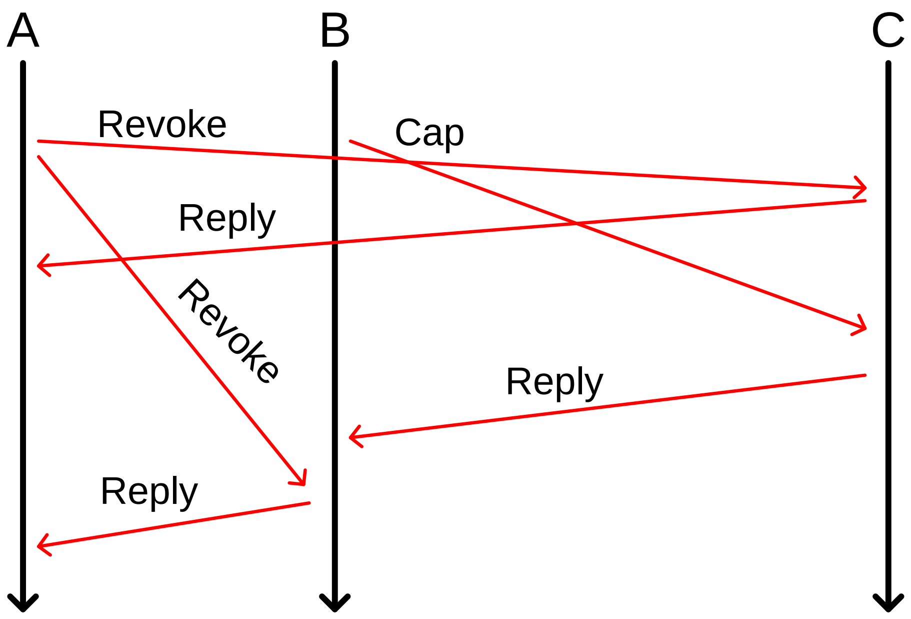


Figure 6.1: Problem with using acknowledments

Delete and transfer: Delete will delete one capability and perform a check for copies. The in-flight capability already has a copy that can satisfy the requirements of delete. If the capability is also being deleted, it is the same two deletes overlapping which indeed is a conflict.

Retype and delete: A delete will not conflict with retypes because it checks for the existence of copies which a successful retype cannot possibly create. A retype will not conflict with deletes because no matter when it performs the check for descendants, it will either see descendants that are about to be deleted or not see them at all, either of the outcomes is correct.

Retype and transfer: Retype cannot conflict with transfers because in order to transfer a capability because the capability being transferred is a copy of an existing capability which does not change the existing descendants relationships.

6.1.3 Incorrect solution

Since the essential issue with the relation with cross-core transfer operation is that in-flight capabilities do not exist anywhere, the sending core can maintain a list of in-flight capabilities which are garbage collected when the receiver acknowledges the reception of the capability. This allows the sending core to include the in-flight capabilities in the search for copies and descendants. As shown below, this intuitive solution is actually incorrect.

Consider the scenario shown in figure **??**. The system consists of three cores {A, B, C}, A is trying to revoke a capability and B is trying to send a copy of the capability to C. A sends a revoke request to B, C and waits for their acknowledgments. B sends the capability to C and adds it to its list of in-flight capabilities. C sees the revoke request first, performs the operation locally and replies to A. Then it sees the capability sent from B, inserts it locally and send an acknowledgment to B. B sees the acknowledgment from C first garbage collecting its list of in-flight capabilities and then sees the revoke request from A, performs the revoke locally, and replies to A. A receives replies from both B and C concluding the revoke operation. The revoke operation has finished and C still has a copy of the revoked capability.

6.2 Requirements from a correct solution

A correct solution will enforce the following transactional like and ordering properties.

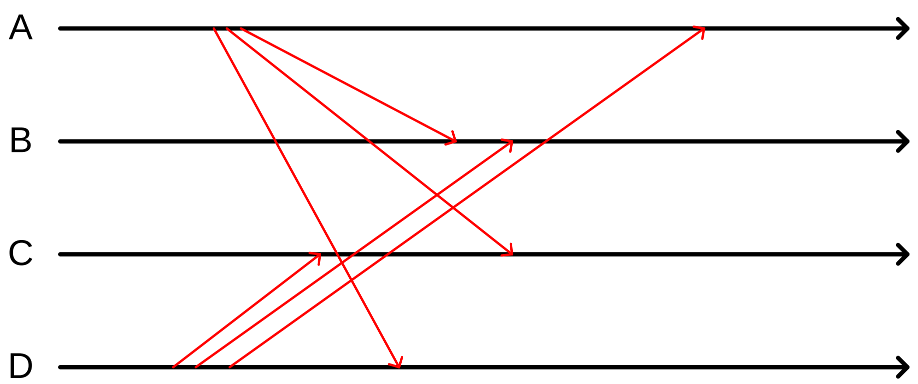


Figure 6.2: Problem with using causal order delivery

6.2.1 Transactional properties

Atomic: If one part of a capability operation fails, then the entire operation must fail. For example, in a retype operation, the descendant capabilities can be created in parallel with the check for existing descendants. If existing descendants are found, then the newly created descendants must be removed.

Consistent: Every capability operation must leave the system in a consistent state. For example, if caching is used an operation deletes the last copy of a capability on a core, then all its remote relations should be informed and updated before the operation finishes.

Isolation: The data that has been modified during a capability operation must not be accessible by other operations till the operation finishes. For example, if a retype operation has eagerly created descendants while checking for existing descendants in parallel, no other capability operation should be able to access the descendants created eagerly till the retype operation finishes.

6.2.2 Ordering

Section **??** discusses the two types of conflicts that can arise when multiple cores try to perform operations on related capabilities at the time. Correctness can be ensured if the individual steps (messages) in operations must be executed in some order. We provide some background on ordering messages before discussing the actual ordering requirements.

Background on ordering: Four types of ordering are possible. We discuss them from the cheapest to the most expensive.

- No order: Messages can be delivered in any order. This cannot happen on Barrelfish.
- Single Source FIFO (SSF): Messages from the same sender are delivered in the order they were sent. This is what currently is available on Barrelfish by default.
- Causal: Messages are delivered based on their partial order given by the happens-before relationship. Vector clocks [?] can be used to provide causal delivery of messages.
- Total order: All receivers receive messages in the same order. Note that any ordering function can be used to order the messages. Further, total order does not imply causal order but if the ordering function respects SSF, then total order does imply causal order.

6.2.3 Conflicts with each-other

The minimum ordering required for resolving conflicts between retype, delete, and revoke operations is total order. Figure **??** illustrates the reason causal ordering does not work. Nodes $\{A,B,C,D\}$ each contain copies of the same capability that $\{A,D\}$ wish to retype at the same time. B receives the message from A before the message from

D, and C receives the D's message before A's. The messages were delivered in causal order, C will refuse to A, B will refuse to D not allowing either retype operation to succeed. Similar examples can be constructed for the delete and revoke operations as well.

Total order delivery will deliver the messages to B and C in the same order allowing just one and only one retype to succeed.

6.2.4 Conflict between revoke and transfer

Figure **??** illustrates the conflict between revoke and cross-core transfer operation. C sees the message from A before the message from B. Hence any message it sends to B causally depend upon the message from A. If this was reflected in the message C sent to B, then B could delay handling the message till it sees the revoke request from A. When B receives the revoke request from A, it will realize that A is trying to revoke an in-flight capability and take appropriate actions to resolve the issue. The delaying of the message can be guaranteed by enforcing causal delivery of messages.

6.2.5 Conflict between revoke and retype

The minimum ordering requirement is total. With less stricter ordering, different cores might see the revoke request and retype requests in different orders. The retyping core might delete the source capability but end up creating descendants later.

6.3 Solutions

We will discuss and compare the following four solutions.

- Locks
- Total order broadcast
- Two phase commit
- Sequencer

6.3.1 Locks

Locks can be used to enforce the above transactional and ordering policies. Critical sections can be used to resolve the conflicts ensuring that *Time-of-check-to-time-of-use* is not violated.

Below, we provide pseudo-code for how capability operations will be implemented when using a centralized lock and not caching information about remote relations.

Copying a capability: This operation is safe to be performed without holding a lock.

```
cap_copy(struct cte *cte) {
  create_copies();
}
```

Retyping a capability: Holding the lock is required to prevent multiple cores from trying to retype copies of the same capability. Acquiring the lock is a blocking call during which the capability might have been deleted so it is important to check that the capability still exists after the lock is acquired.

```
cap_retype(struct cte *cte) {
  errval_t err = success;
  acquire_lock();
  if (cap_exists(cte) == false) {
    err = fail;
    goto done;
  }
  if (has_descendants(cte) == true) {
    err = fail;
    goto done;
  }
  create_descendants(cte);
done:
  release_lock();
  return err;
}
```

Deleting a capability: If the capability has local copies, the operation is purely local. Otherwise, holding the lock is required to ensure that if the last copy of the capability in the system is being deleted, then the type-specific cleanup operations are executed.

```
cap_delete(struct cte *cte) {
  remove_cap(cte);

  if (!requires_typed_ops(cte)) {
    return success;
  }
  if (has_local_copies(cte)) {
    return success;
  }

  errval_t err = success;
  acquire_lock();
  if (!cap_exists(cte) {
    goto done;
  }

  if (!has_copies(cte) {
    type_specific_ops(cte);
  }

done:
  release_lock();
  return err;
}
```

Revoking a capability: Holding the lock is required to ensure that if multiple cores are trying to revoke related capabilities, only one succeeds. This is why the code below ensures that the capability still exists after acquiring the lock.

```
cap_revoke(struct cte *cte) {
  errval_t err = success;
  acquire_lock();
  if (!cap_exists(cte) {
    err = fail;
    goto done;
  }

  while(has_descendants(cte) {
    struct cte *dest = get_next_descendant(cte);
    remove_cap(dest);
  }
  while(has_copies(cte) {
    struct cte *dest = get_next_copy(cte);
    remove_cap(dest);
  }

done:
  release_lock();
  return err;
}
```

Cross-core transferring a capability: Holding the lock is required to conflicts between the above capability operations and cross-core transfer operation. The sender acquires the lock and sends the capability. The receiver creates the capability and releases the lock.

```
cap_send(struct cte *cte, coreid_t dest) {
  acquire_lock();
  if (cap_exists(cte) == false) {
    return fail;
  }
  send_cap(cte, dest);
}

cap_receive(struct cte *cte) {
  create_cap(cte);
  release_lock();
  return_success_to_app();
}
```

Caching remote relations

If remote relations are cached, then the cache has to be kept consistent when capability operations change the state of relations. Below we describe the modifications that will be required in the above pseudo-code to keep the cache consistent. Note that our cache can be seen as eager replication. As discussed in section **??**, we will be using the mark all approach to maintain the cache.

Copying a capability: When the new copy is created, its cache of remote relations is set equal to the cache from the capability it was copied from.

Retyping a capability: If the capability does not have remote copies and descendants, the operation is purely local and the lock is not required. If the operation is not local, then the lock is acquired, checks for existence of capability and for descendants and is made, the capability is retyped, and remote relations are updated based on the rules presented in section **??**.

```
cap_retype(struct cte *cte) {
  if (has_local_descendants(cte) == true) {
    return fail;
  }

  if (!has_remote_copies(cte) && !has_remote_descendants(cte)) {
    create_descendants(cte);
    return success;
  }

  if (has_remote_descendants(cte)) {
    return fail;
  }
  errval_t err = success;
  acquire_lock();
  if (!cap_exists(cte)) {
    err = fail;
    goto done;
  }

  if (has_remote_descendants(cte)) {
    err = fail;
    goto done;
  }
  if (has_local_descendants(cte)) {
    err = fail;
    goto done;
  }

  create_descendants(cte);
  update_relations(cte);

done:
  release_lock();
  return err;
}
```

Deleting a capability: If the capability has local copies or no remote relations, the operation is purely local. Otherwise, the lock is required and the remote relations must be updated.

```
cap_delete(struct cte *cte) {
  remove_cap(cte);

  if (!requires_typed_ops(cte)) {
    return success;
  }
  if (has_local_copies(cte)) {
    return success;
  }

  if (!has_remote_relations(cte)) {
    type_specific_ops(cte);
    return success;
  }

  acquire_lock();
  if (!cap_exists(cte)) {
    goto done;
  }
  if (!has_copies(cte) {
    type_specific_ops(cte);
  }
  update_remote_relations(cte);
  remove_cap(cte);
  release_lock();
done:
  return success;
}
```

Revoking a capability: If the capability has no remote relations, the operation is purely local. Otherwise, the lock is required.

```
cap_revoke(struct cte *cte) {
  if (!has_remote_relations(cte)) {
    while(has_descendants(cte) == true) {
      struct cte *dest = get_next_descendant(cte);
      remove_cap(dest);
    }
    while(has_copies(cte) == true) {
      struct cte *dest = get_next_copy(cte);
      remove_cap(dest);
    }
    return success;
  }

  errval_t err = success;
  acquire_lock();
  if (!cap_exists(cte)) {
    err = fail;
    goto done;
  }

  while(has_descendants(cte) == true) {
    struct cte *dest = get_next_descendant(cte);
    remote_cap(dest);
  }
  while(has_copies(cte) == true) {
    struct cte *dest = get_next_copy(cte);
    remote_cap(dest);
  }
  release_lock();
done:
  return err;
}
```

Cross-core transferring a capability: The remote relations cache on local and remote capabilities is updated as presented in section **??**.

Multiple locks

[*NYI*] If a single lock for the entire system is used, only one capability operation can be performed at any given moment limiting the available potential for parallelism. By using different locks for unrelated capabilities, multiple capability operations can proceed in parallel. Using multiple locks will increase the space requirement but will also improve parallelism.

[*Multiple locks are not straight-forward. Ancestors reference more memory than descendants do.*]

[*Can a copy lock for delete, a descendant lock for retype, and both for revoke work?*]

6.3.2 Total order broadcast

[*Use a single data structure for all pending cap operations.*]

The required transactional and ordering guarantees can be provided by ensuring that messages for related capabilities are delivered in the same order on all cores. Note that causal order delivery resolves the conflict between retype, delete, revoke and cross-core transfer operation but not within the retype, delete, revoke operations.

Below we present pseudo-code for how the capability operations will be implemented when not caching information about remote relations.

Copying a capability: This operation is safe to be performed without any ordering requirements.

Deleting a capability: If the capability does not require type-specific operations or has local copies, the operation succeeds. Or else, local state is created and a message is broadcast to all cores.

```
cap_delete(struct cte *cte) {
  if (!requires_typed_ops(cte) || has_local_copies(cte)) {
    remove_cap(data->cte);
    return success;
  }

  struct delete_data *data = malloc(sizeof(struct delete_data));
  delete_data_initialize(data, cte);
  outstanding_delete_list->add(data);
  send_delete_request_bcast(TOTAL_ORDER, data);
}
```

If a core receives a delete request broadcast and it was the sender of the message, it removes the capability and returns. If the core was not the sender of the message, then it replies with the state of local copies of the specified capability.

```
delete_request_bcast(coreid_t from, struct delete_request *data) {
  if (from == my_core_id) {
    remove_cap(data->cte);
    return;
  }

  send_delete_reply(from, data, has_local_copies(data->cte));
}
```

Finally, when a core receives a reply from a delete request, if a remote copy is found, no type-specific cleanup is required and the operation finishes. If all replies have been aggregated and no copies were found, then the type-specific cleanups are performed and then the operation finishes.

```
delete_reply(coreid_t from, bool has_copies) {
  struct cte *my_data = outstanding_deletes_list->get(data);
  if (!my_data) {
    return;
  }

  if (has_copies) {
    outstanding_deletes_list->remove(my_data);
    return;
  }

  increment_replies(my_data);
  if (seen_all_replies(my_data)) {
    type_specific_ops(cte);
    outstanding_deletes_list->remove(my_data);
  }
}
```

Since the deleting cores remove the capability when they receive the broadcast, when multiple cores are trying to delete copies, the last core's broadcast will see no copies while other cores will see copies.

[*Malloc probably means unbounded memory requirement.*]

Retyping a capability: If the capability has local descendants, then the operation fails. Else, a retype request broadcast is sent.

```
cap_retype(struct cte *cte) {
  if (has_local_descendants(cte)) {
    return fail;
  }

  struct retype_data *data = malloc(sizeof(struct retype_data));
  retype_data_initialize(data, cte);
  outstanding_retypes_list->add(data);
  send_retype_request_bcast(data);
}
```

When a core receives a retype request broadcast and it was the sender of the message, one of two things may have happened. Either the core had already received a broadcast from another core trying to retype the same capability in which case the receiver's operation has failed and the state for the request has been removed or the core can succeed in retyping the capability and updates its state accordingly. If the core was not the sender of the broadcast, it sends a reply to the sender with the state of its local descendants. Then the core checks if it has an outstanding retype request for the capability. If it does and its request has not been delivered yet, its retype fails. An error is sent to the application and the state is garbage collected.

```
retype_request_bcast(coreid_t from, struct retype_request *data) {
  if (from == my_core_id) {
    struct cte *my_data = outstanding_retypes_list->get(data);
    if (!my_data) {
      return;
    }
    my_data->can_succeed_flag = true;
    return;
  }

  send_retype_reply(from, data, has_local_descendants(data->cte));
  struct cte *my_data = outstanding_retypes_list->get(data);
  if (!my_data) {
    return;
  }

  if (!my_data->success_flag) {
    outstanding_retypes_list->remove(my_data);
    return_failure_to_app();
  }
}
```

When a core receives a reply to the retype request broadcast, the operation may already have been failed, in which case the reply is ignored. If the operation has not been canceled yet, then if the reply indicates no descendants, the operation can still succeed. If all replies are seen then the retype operation succeeds.

```
retype_reply(struct retype_request *data, bool has_descendants) {
  struct cte *my_data = outstanding_retypes_list->get(data);
  if (!my_data) {
    return;
  }

  if (has_descendants) {
    outstanding_retypes_list->remove(my_data);
    return_failure_to_app();
    return;
  }

  increment_replies(my_data);
  if (seen_all_replies(my_data)) {
    create_descendants();
    outstanding_retypes_list->remove(my_data);
    return_success_to_app();
  }
}
```

This resolves the conflicts between two retypes. The conflict between retype and revoke are discussed below.

[*Malloc probably means unbounded memory requirement.*]

Revoking a capability: The core initializes some local state and then broadcasts a revoke request to all cores in the system.

```
cap_revoke(struct cte *cte) {
  struct revoke_data *data = malloc(sizeof(struct revoke_data));
  revoke_data_initialize(data, cte);
  outstanding_revokes_list->add(data);
  send_revoke_request_bcast(TOTAL_ORDER, data);
}
```

When a core receives a revoke request broadcast, if the core was trying to retype a related capability, then it fails the retype operation. If it is not trying to revoke the capability itself, it simply revokes the capability locally and sends a reply. If the core is trying to revoke the same capability but it was not the sender of the broadcast, then this core's revoke operation fails.

```
revoke_request_bcast(coreid_t from, struct revoke_request *data) {

  if (related_retype_in_progress(data->cte)) {
    outstanding_retypes_list->remove(data);
    return_fail_to_app();
  }

  struct cte *my_data = outstanding_revokes_list->get(data);
  if (!my_data) {
    revoke_locally(data->cte);
    send_revoke_reply(from, data);
    return;
  }

  if (from != my_core_id && !my_data->success_flag) {
    outstanding_revokes_list->remove(my_data);
    send_revoke_reply(from, data);
    return_failure_to_app();
    return;
  }

  my_data->success_flag = true;
}
```

When a core receives a reply from the broadcast, it aggregates them till it has heard back from all cores in the system and then sends a success to the application.

```
revoke_reply(struct revoke_request *data) {
  struct cte *my_data = outstanding_revokes_list->get(data);
  if (!my_data) {
    return;
  }

  increment_replies(my_data);
  if (seen_all_replies(my_data) && !my_data->sent_transfer_cap_delete_flag) {
    outstanding_revokes_list->remove(my_data);
    return_success_to_app();
  }
}
```

cross core transfer: When sending a capability to another core, the sending core creates local states and broadcasts the send message to all cores in the system.

```
cap_transfer(struct cte *cte, coreid_t to) {
  struct transfer_data *data = malloc(sizeof(struct transfer_data));
  transfer_data_initialize(data, cte);
  outstanding_transfers_list->add(data);
  send_transfer_request_bcast(TOTAL_ORDER, data, to);
}
```

When a core receives the capability transfer broadcast, it checks against the state of current capability operations in progress and takes appropriate actions if they are related.

If a revoke operation is in progress that is trying to revoke a copy or an ancestor of the capability being transferred, the broadcast will indicate that the system indeed has more copies and descendant of the capability being revoked which must be deleted. If the receiver was also the recipient of the capability, it does not create it and returns an error to the sender of the capability or else the receiver sends a message to the core to which the capability is being transferred to requesting it to delete the capability.

```
transfer_request_bcast(coreid_t from, struct transfer_data *data, coreid_t to) {
  struct cte *my_data;
  my_data = outstanding_revokes_list->get(data);
  if (my_data) {
    if (to == my_core_id) {
      send_transfer_reply(from, FAILURE_REVOKED, data);
      return;
    }

    my_data->sent_transfer_cap_delete_flag = true;
    send_delete_transferred_cap_request(to, data);
  }

  if (to != my_core_id) {
    return;
  }

  my_data = pending_delete_for_transferred_cap_list->get(data);
  if (my_data) {
    pending_delete_for_transferred_cap_list->remove(my_data);
    send_delete_transferred_cap_reply(my_data->from);
    send_transfer_reply(from, FAILURE_REVOKED, data);
  }

  cap_create(data->cte);
  send_transfer_reply(from, SUCCESS);
}
```

When the sender of the capability gets a reply from the receiver, it forwards the error code to the application and garbage collects its state.

```
transfer_reply(errval_t err, struct transfer_request *data) {
  my_data = outstanding_transfers_list->get(data);
  outstanding_transfers_list->remove(data);
  return_err_to_app(my_data->app, err);
}
```

If a revoke was in progress during the transfer of the capability, the core revoking the capability sends a message to the receiver of the transferred capability to delete it. When the receiver receives this message, it may or may not have received the capability yet. If it has already received the capability, it deletes or else it establishes some state to delete when it is later received.

```
delete_transferred_cap_request(coreid_t from, struct data *data) {
  if (cap_exists(data->cte)) {
    remove_cap(cte);
    send_delete_transferred_cap_reply(from);
    return;
  }

  struct pending_delete_for_transferred_cap *my_data =
    malloc(sizeof(struct pending_delete_for_transferred_cap));
  pending_delete_for_transferred_cap_initialize(my_data);
  pending_delete_for_transferred_cap_list->add(my_data);
}
```

```
delete_transferred_cap_reply(struct data *data) {
  my_data = outstanding_revokes_list->get(data);
  if (my_data) {
    return;
  }

  my_data->sent_transfer_cap_delete_flag = false;
  if (seen_all_replies(my_data)) {
    outstanding_revokes_list->remove(my_data);
    return_success_to_app();
  }
}
```

[*Caching NYI*]

6.3.3 Two phase commit

[*NYI*] Use 2pc.

6.3.4 Sequencer

[*NYI*] Use a sequencer. This will order whole operations.

6.3.5 Comparison

Compare the approaches

Chapter 7

Implementation details

- Using THC
- Using RPC between app and monitor
- Using RPC between monitors
- Everything having ancestors on memserv
- Optimization: monitor caches root cnodes
- Bootstrap
- How to put enum objtype in interface file?
- Two types of ifdefs: type of caching (none, list, or bits) and type of mgmt

7.1 Performing capability operations

If caching is not used, then the application knows for which operations it must contact the monitor and does so directly without requesting the kernel to try to perform the operation first.

If caching is used, then the application should try to perform the operation via the kernel first. If the capability does have remote relations, the kernel returns an appropriate error to the application in which case it contacts the monitor.

7.2 Sharing mdb between kernel and monitor

When the application wants the monitor to perform an operation for it, it passes the monitor its root CNode and all the required parameters.

Chapter 8

Not Yet Discussed

Things that I know that I need to discuss.

- The OS does not guarantee which order the operations will be performed in. The user must enforce the ordering herself.
- Partitioned approach with caching is eager replication. Consider lazy replication.