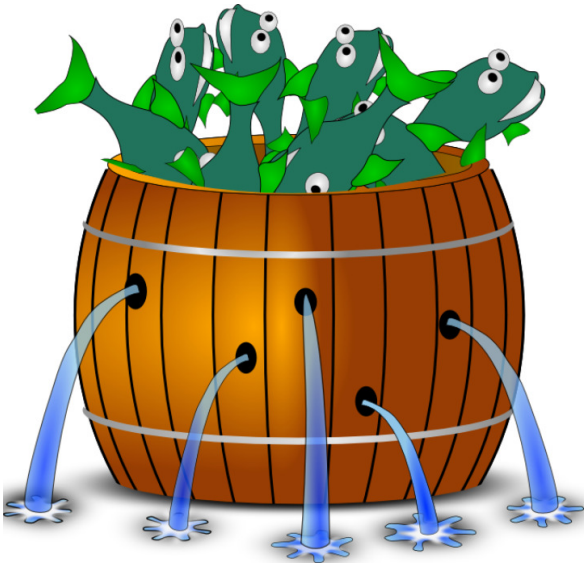
Barrelfish Project
ETH Zurich



Inter-dispatcher communication in Barrelfish

Barrelfish Technical Note 011

Andrew Baumann

05.12.2011

Systems Group
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
0.0	06.08.2010	AB	Initial version
0.1	01.09.2010	AB	Documented Flounder IDL, stub API and waitset
0.2	21.10.2010	AB	Completed runtime support chapter
0.3	05.12.2011	MN	Added meta-parameter note, AHCI references

Contents

Chapter 1

Introduction

TODO

Chapter 2

Flounder stub compiler

2.1 Overview

2.2 Interface definition language

2.2.1 Lexical conventions

The following convention, derived from Mackerel [1] has been adopted for Flounder. It is similar to the convention in modern day programming languages like C and Java.

Whitespace: As in C and Java, Flounder considers sequences of space, newline, tab, and carriage return characters to be whitespace. Whitespace is generally not significant.

Comments: Flounder supports C-style comments. Single line comments start with `//` and continue until the end of the line. Multiline comments are enclosed between `/*` and `*/`; anything inbetween is ignored and treated as white space.

Identifiers: Valid Flounder identifiers are sequences of numbers (0-9), letters (a-z, A-Z) and the underscore character "_". They must start with a letter or "_".

$$\begin{aligned} identifier &\rightarrow (letter \mid \_)(letter \mid digit \mid \_) \\ letter &\rightarrow (\mathsf{A} \ldots \mathsf{Z} \mid \mathsf{a} \ldots \mathsf{z}) \\ digit &\rightarrow (\mathsf{0} \ldots \mathsf{9}) \end{aligned}$$

Integer Literals: A Flounder integer literals is a sequence of digits, optionally preceded by a radix specifier. As in C, decimal (base 10) literals have no specifier and hexadecimal literals start with `0x`.

$$\begin{aligned} digit &\rightarrow (\mathsf{0} \ldots \mathsf{9}) \\ hexadecimal &\rightarrow (\mathsf{0x})(\mathsf{0} \ldots \mathsf{9} \mid \mathsf{A} \ldots \mathsf{F} \mid \mathsf{a} \ldots \mathsf{f}) \end{aligned}$$

Reserved words: The following are reserved words in Flounder:

```
alias     call      enum      in        interface  message
out       response  rpc       struct    typedef
```

Builtin types: The following identifiers are recognised as builtin types:

```
bool          cap           char          give_away_cap int           int8
int16         int32         int64         intptr        iref          size
string        uint8         uint16        uint32        uint64        uintptr
```

Special characters: The following characters are used as operators, separators, terminators or for other special purposes in Flounder:

```
{ } [ ] ( ) ; ,
```

2.2.2 Interface specification

A Flounder input file must specify a single interface, of the form:

```
interface name ["description"]
{
  types;
  ...
  messages;
  ...
};
```

name is an identifier for the interface type, and will be used to generate identifiers in the target language (typically C). While not currently enforced by the Flounder compiler, by convention this should only differ from the name of the file in case and suffix, e.g. the file `xapic.dev` defines a device type of name `xAPIC`.

description is a string literal in double quotes, which describes the interface being specified, for example `"Name service"`.

After the description string follows in curly brackets a series of declarations which specify types, messages and their arguments. Each declaration must be one of the following:

- A transparent type alias, using the `alias` declaration, described in Section 2.2.3.
- A type definition, using the `typedef` declaration, described in Section 2.2.4.
- A simple message specification, using either the `message`, `call` or `response` declarations, describing a single unidirectional message with typed arguments, and described in Section 2.2.5.
- An `rpc` specification, which describes two related messages (a call and the response) with a single declaration, and described in Section 2.2.6.

Type aliases and definitions (`alias` and `typedef`) are optional, but if present must precede all messages. There must be at least one message in every interface.

2.2.3 Transparent type alias

```
alias aliasname builtintype;
```

This simply declares a given alias name that may henceforth be used in the interface file in place of the given builtin type. It does not alter the generated code.

2.2.4 Type definition

In contrast to an alias, the `typedef` keyword causes a new type to be created in the context of both the interface file and the programming language where the generated stubs are used.

Structure

```
typedef struct {
  typename fieldname;
  ...
} structname;
```

This declares a compound type, akin to a structure in C.

Fixed-size array

```
typedef typename arrayname[size];
```

Where *typename* is a previously-defined type, and *size* is an integer literal, defines an array type with the given *arrayname*. This name refers to an array of a fixed size.

Enumeration

```
typedef enum {
  fieldname,
  ...
} structname;
```

An enumeration, as in C, is represented as an integer type that may take one of the predefined symbolic values.

Alias

```
typedef aliasname typename;
```

This form creates a new name (in both Flounder and C) for a previously-defined or builtin type.

2.2.5 Simple message

A simple message specification takes the form:

```
@context(name=value, ... ) (optional)
message|call|response name (argdecl, ... );
```

The three identifiers `message`, `call`, and `response` exist for legacy reasons, and are treated identically by the stub compiler. They may be used as a simple form of documentation, when messages are intended to travel in a particular direction between client and server.

The meta-parameters beginning with the context identifier are optional and may be used by individual stub compilers for additional information about messages that is not part of the payload. Multiple such contexts may also be provided.

Message arguments use a comma-separated C-like syntax, and may take one of the following forms:

Simple arguments are denoted by type name and argument name literals, separated by whitespace.

Dynamic array arguments include the type name, argument name, and size name literals, as follows:

typename argname[*sizename*]

For example, the following defines a message named "dummy" with three arguments: an integer, a character string, and a dynamic array of unsigned 8-bit integers (ie. a data buffer).

```
message dummy(int32 arg, string s, uint8 buf[buflen]);
```

2.2.6 RPC

@*context*(*name*=*value*, ...) *(optional)* rpc *name* (in|out *argdecl*, ...);

An `rpc` declaration defines a pair of related one-way messages for the RPC *call* and its *response*. Argument declarations take the same syntax as for a simple message, but are prefixed with either `in` or `out`. All `in` arguments are arguments of the call, and all `out` arguments are arguments of the response.

For example, the following defines an RPC that takes as input a string and returns an `errval` type (which must previously have been defined):

```
rpc testrpc(in string s, out errval reterr);
```

With the exception of its special treatment by the RPC client backend (described in Section 6.3), this RPC is exactly equivalent to the following message specifications:

```
message testrpc_call(string s);
message testrpc_response(errval reterr);
```

2.3 Invocation

When invoked, Flounder processes a single interface file, and generates into a single output file the results of one or more *backends*. The names of the input and output files and the backends to use are determined by the command-line arguments, which are as follows:

```
Usage: flounder [OPTION...] input.if output
  -G       --generic-header    Create a generic header file
           --generic-stub      Create generic part of stub implemention
  -a ARCH  --arch=ARCH         Architecture for stubs
  -i FILE  --import=FILE       Include a given file before processing
           --lmp-header        Create a header file for LMP
           --lmp-stub          Create a stub file for LMP
           --ump-header        Create a header file for UMP
           --ump-stub          Create a stub file for UMP
           --rck-header        Create a header file for RCK
           --rck-stub          Create a stub file for RCK
           --bmp-header        Create a header file for BMP
           --bmp-stub          Create a stub file for BMP
           --loopback-header   Create a header file for loopback
           --loopback-stub     Create a stub file for loopback
```

```
          --rpcclient-header  Create a header file for RPC
          --rpcclient-stub    Create a stub file for RPC
          --msgbuf-header     Create a header file for message buffers
          --msgbuf-stub       Create a stub file for message buffers
  -T      --thc-header        Create a THC header file
  -B      --thc-stubs         Create a THC stubs C file
          --ahci-header       Create a header file for AHCI
          --ahci-stub         Create a stub file for AHCI
```

Zero or more files may be specified as *imports*. An import file provides a simplistic mechanism for sharing type definitions among multiple interfaces: it consists of one or more `alias` or `typedef` statements. All import files are processed in the order in which they occur on the command line, before parsing of the interface definition file begins. Otherwise, types defined in import files are equivalent to types appearing in an interface definition.

The architecture option must be specified when generating stubs for either LMP, UMP, or RCK backends; when specified, it must be one of the architectures Flounder understands (see `Arch.hs` in the flounder source), which is typically the same as the set of architectures supported by Barrelfish.

All other options cause Flounder to emit the C code generated by one of its backends. The *generic* backend defines the common interface and support code that is shared by all other backends, and is always required. Chapter 8 describes the location and contents of the generated files that are used in the Barrelfish build system.

Chapter 3

Flounder stub API

Most Flounder backends generate message stubs for specific interconnect drivers, with implementations varying widely according to the properties of the interconnect driver. However, common to all interconnect drivers for a given interface is the interface to the *binding object*, which allows the typed messages defined in the interface definition to be sent and received in an event-driven programming style. The interface to the binding object for a given interface is generated by the `--generic-header` option to Flounder, and is documented here.

Most interaction with bindings involves interface-type-specific functions. In the examples that follow, we assume that a Flounder interface named `iface` is used, with the following specification:

```
interface iface {
  message basic(uint32 arg);
  message str(uint32 arg, string s);
  message caps(uint32 arg, cap cap1, cap cap2);
  message buf(uint8 buf[buflen]);
};
```

3.1 Binding process

In order to communicate, two dispatchers must establish a communication channel using a common interconnect driver, and must acquire binding objects for each endpoint of the channel. This process is known as *binding*, and operates in two modes:

1. A *client* dispatcher initiates a binding to a service by calling the interface's *bind* function
2. A *server* dispatcher accepts an incoming binding request from a client, on a service which it has previously *exported*

In any binding attempt, one dispatcher must act as server and the other as the client, however once a binding is established, the binding objects and the process of communication on both sides of the binding are equivalent and indistinguishable.

3.1.1 Server-side: export

In order to accept binding requests, a dispatcher must first export a service. This is done by calling the *export* function for the given interface:

```
typedef void idc_export_callback_fn(void *st, errval_t err, iref_t iref);
typedef errval_t iface_connect_fn(void *st, struct iface_binding *binding);
```

```
errval_t iface_export(void *st, idc_export_callback_fn *export_cb,
                      iface_connect_fn *connect_cb, struct waitset *ws,
                      idc_export_flags_t flags);
```

The `iface_export()` function initiates the export of a new service. It takes a state pointer, export callback, connection callback, waitset pointer, and flags. If it returns success, the export has been successfully initiated.

When the export either succeeds or fails, the export callback will be invoked. It takes the state pointer (which was passed to `iface_export()`), an error code indicating success or failure of the export, and an *interface reference* (iref). The iref is only meaningful if the export succeeds; if this is the case, it identifies the exported service in such a way that a client can use the iref to initiate a binding to the service. In order for the service to be usable, the iref must be communicated to the potential clients in some manner; typically by registering it in the name service, which is described in Chapter 7.

Once a service has been exported, clients may attempt to bind to it. When this occurs, the connection callback will be invoked. This function takes the state pointer used at export time, and a pointer to the new partly-constructed binding object. If it returns success, the binding request will be accepted, and the binding process will complete. If it returns an error code indicating failure, the binding attempt will be rejected, and the error code will be returned to the client. When accepting a binding request, the connection callback also has the task of filling in the vtable of receive handers, as described in Section 3.3, and the error handler, described in Section 3.6.

3.1.2 Client-side: bind

In order to initiate a binding, a client dispatcher calls the *bind* function for a given interface:

```
typedef void iface_bind_continuation_fn(void *st, errval_t err,
                                        struct iface_binding *binding);

errval_t iface_bind(iref_t i, iface_bind_continuation_fn *continuation,
                    void *st, struct waitset *waitset, idc_bind_flags_t flags);
```

This function takes the iref for the service (obtained, for example, by a name service query), a bind continuation function, state pointer, waitset poitner, and flags. If it returns success, a binding attempt has been initiated.

At some later time, when the binding either completes or fails, the bind continuation function will be run. This takes the state pointer passed to `iface_bind()`, an error code indicating success or failure, and a pointer to the newly-constructed binding, which is only valid if the error code indicates success.

3.2 Binding objects

The end result of the binding process (either client-side or server-side) is a binding object, which is the abstract interface to a specific interconnect driver for a specific interface type. It is implemented in C as the `struct iface_binding` type, the defined portion of which is shown below, and described in the following sections:

```
struct iface_binding {
    /* Arbitrary user state pointer */
    void *st;

    /* Waitset used for receive handlers and send continuations */
    struct waitset *waitset;

    /* Mutex for the use of user code. */
```

```
    /* Must be held before any operation where there is a possibility of
     * concurrent access to the same binding (eg. multiple threads, or
     * asynchronous event handlers that use the same binding object). */
    struct event_mutex mutex;

    /* returns true iff a message could currently be accepted by the binding */
    iface_can_send_fn *can_send;

    /* register an event for when a message is likely to be able to be sent */
    iface_register_send_fn *register_send;

    /* change the waitset used by a binding */
    iface_change_waitset_fn *change_waitset;

    /* perform control operations */
    iface_control_fn *control;

    /* error handler for any async errors associated with this binding */
    /* must be filled-in by user */
    iface_error_handler_fn *error_handler;

    /* Incoming message handlers (filled in by user) */
    struct iface_rx_vtbl rx_vtbl;
};
```

The `st` pointer is available for users to associate arbitrary local state with each binding object. It is not used by the binding implementation.

The `waitset` pointer identifies the waitset used by the binding for executing receive handler functions, send continuations, and internal logic. It is read-only for users of the binding; to change the actual waitset, the `change_waitset()` control function must be used.

Binding implementations are not thread-safe, and it is the responsibility of user code to ensure that no concurrent access to the same binding occurs. The `mutex` field is not used by the binding implementation, but is provided for user code to ensure safety of concurrent accesses to the binding object. It is used, for example, to arbitrate access to the shared monitor binding, as described in Section 5.1.1.

The remaining fields are documented in the following sections.

3.3 Receiving messages

When a message arrives, the binding implementation calls the *receive handler* function located in the binding's `rx_vtbl` table of function pointers. This table has an entry for each message type, with arguments corresponding to those of the message, as shown below:

```
/*
 * Message type signatures (receive)
 */
typedef void iface_basic__rx_method_fn(struct iface_binding *_binding, uint32_t arg);
typedef void iface_str__rx_method_fn(struct iface_binding *_binding, uint32_t arg, char *s);
typedef void iface_caps__rx_method_fn(struct iface_binding *_binding, uint32_t arg,
                                      struct capref cap1, struct capref cap2);
typedef void iface_buf__rx_method_fn(struct iface_binding *_binding, uint8_t *buf, size_t buflen);

/*
 * VTable struct definition for the interface (receive)
 */
struct iface_rx_vtbl {
    iface_basic__rx_method_fn *basic;
```

```
    iface_str__rx_method_fn *str;
    iface_caps__rx_method_fn *caps;
    iface_buf__rx_method_fn *buf;
};
```

It is the responsibility of the user to fill in the `rx_vtbl` when the binding is initialised, either in the connection callback on the server side, or in the bind callback on the client side. This is typically achieved by copying a static table of function pointers. It is a fatal error if a message arrives on a binding and the handler function for that message type is uninitialised or NULL.

Any message arguments passed by reference become the property of the user, and must be freed by the appropriate memory management mechanisms:

- Strings and arrays live on the heap, and should be released by a call to `free()`
- Capabilities refer to an occupied capability slot allocated from the standard capability space allocator, and should be released by a call to `cap_destroy()`

3.4 Sending messages

A message may be sent on the binding by calling the appropriate transmit function. For our example interface, these are as follows:

```
errval_t iface_basic__tx(struct iface_binding *_binding, struct event_closure _continuation,
                         uint32_t arg);
errval_t iface_str__tx(struct iface_binding *_binding, struct event_closure _continuation,
                       uint32_t arg, const char *s);
errval_t iface_caps__tx(struct iface_binding *_binding, struct event_closure _continuation,
                        uint32_t arg, struct capref cap1, struct capref cap2);
errval_t iface_buf__tx(struct iface_binding *_binding, struct event_closure _continuation,
                       const uint8_t *buf, size_t buflen);
```

Each function takes as arguments the binding pointer and a *continuation* closure (described below), followed by the message payload. If it returns success, the message has been successfully enqueued for transmission, and will eventually be sent if no asynchronous errors occur.

The continuation is optional. When specified, it provides a generic closure which will be executed after the given message has been successfully sent. Two macros are provided to construct continuations: `NOP_CONT` provides a no-op continuation, to be used when this functionality is not required, whereas `MKCONT(func,arg)` constructs a one-off continuation given a function pointer and its argument.

All binding implementations can buffer exactly one outgoing message. If another send function is called on the same binding while the buffer is full, the `FLOUNDER_ERR_TX_BUSY` error will be returned, indicating that the binding is not currently able to accept another message. In this case, the user must arrange to retry the send at a later time, typically from a send continuation function, or by using the `register_send()` control function (described below).

Any message arguments passed by reference (strings, arrays, and capabilities) are borrowed by the binding, and must remain live for as long as the message is buffered but not yet completely sent. The send continuation may be used to determine when it is safe to reclaim the memory used by reference parameters.

3.5 Control functions

Four standard control functions are defined for all binding implementations, and are invoked through function pointers in the binding object: `can_send`, `register_send`, `change_waitset` and `control`.

can_send

```
typedef bool iface_can_send_fn(struct iface_binding *binding);
```

This function returns true if and only if the binding could currently accept an outgoing message for transmission. Note this does not indicate the message could immediately be delivered to the underlying channel (further queuing may be involved), simply that a send function would succeed.

register_send

```
typedef errval_t iface_register_send_fn(struct iface_binding *binding,
                        struct waitset *ws, struct event_closure continuation);
```

This function arranges for the given continuation closure to be invoked on the given waitset when the binding is able to accept another outgoing message (i.e. when `can_send` would next return true). It may fail if a continuation is already registered and has not yet executed; only one continuation may be registered at a time.

change_waitset

```
typedef errval_t iface_change_waitset_fn(struct iface_binding *binding, struct waitset *ws);
```

This function changes the waitset used by the binding for send continuations and running its internal event handlers.

control

```
typedef errval_t iface_control_fn(struct iface_binding *binding, idc_control_t control);
```

This function provides a generic control mechanism. The currently-defined control operations are:

`IDC_CONTROL_TEARDOWN` Initiate connection teardown.

`IDC_CONTROL_SET_SYNC` Enable synchronous optimisations. See below.

`IDC_CONTROL_CLEAR_SYNC` Disable synchronous optimisations. Currently only implemented for the LMP interconnect driver, this operation disables the (default) behaviour of yielding the CPU timeslice to the receiver on every message send.

3.6 Error handling

The binding structure contains an `error_handler` function pointer field, of the following type:

```
typedef void iface_error_handler_fn(struct iface_binding *binding, errval_t err);
```

This function is called to report any asynchronous errors occuring on the binding, including connection teardown, and should be filled-in by the user during the binding process (i.e. whenever the `rx_vtbl` is installed). The default implementation of the error handler for every binding simply prints a message and terminates the current dispatcher.

Chapter 4

Runtime support

This chapter describes the IDC runtime support that is part of `libbarrelfish` which is used by the Flounder-generated stubs.

4.1 Waitsets and event handling

At the core of the Barrelfish event handling mechanism lies the *waitset*, represented by the `struct waitset` type in C. Informally, waitsets facilitate the rendezvous of *threads* with *events*, where an event is represented simply as a closure (function and argument pointers), as follows:

```
struct event_closure {
    void (*handler)(void *arg);
    void *arg;
};
```

These events are typically raised by message channels in response to activity, such as the ability to receive a message on a specific channel. In the following we introduce the abstract properties of channels as they interact with the waitset; the concrete channel interfaces are specific to each interconnect driver and are described in the respective parts of Chapter 5.

All waitsets are local to a specific dispatcher, and may not be accessed from other dispatchers. Although there is no limit to the number of waitsets used, there always exists one *default waitset* for each dispatcher, which may be located by calling the following function:

```
struct waitset *get_default_waitset(void);
```

4.1.1 (Abstract) event channels

All events managed by a waitset originate in *channels*,[1] represented internally by the `struct waitset_chanstate` type. This struct is typically embedded into the state of a real message channel or other event mechanism.

Channels are used to track the registration of handlers for and occurrence of a single specific event. Multiple events (such as the ability to send a message, as distinct from the ability to receive a message) are supported by distinct channels – an interconnect driver will typically use multiple event channels in its implementation. Moreover, each event channel supports only one registration – registration of multiple event handlers on a single event may be implemented above this mechanism.

[1] The term "channel" is now something of a misnomer in this context, as events are no longer used exclusively for message channels.

Every channel exists in one of the following states:

Unregistered: the channel is initialised, but no event handler has been registered, and it is not associated with a waitset

Idle: an event handler closure has been registered with the channel on a specific waitset, but the event has not yet occurred

Polled: this is similar to an idle channel, in that an event has been registered and not yet occurred, but due to the nature of the channel implementation, it must be periodically polled for activity to detect the occurrence of the event

Pending: the event for which a handler was registered has occurred, and the channel is now waiting to deliver the event

The following operations on channels are used by interconnect drivers to change their state in the waitset:

Register: Associates an event handler with a previously-unregistered channel, and moves it to either the *idle* or *polled* state

Deregister: Cancels the previously-registered event handler of a channel, moving it from either *idle*, *polled* or *pending* state back to unregistered

Trigger: Signals occurrence of an event on a *idle* or *polled* channel, and changes it to the *pending* state

These operations are available only to interconnect drivers, using the "private" API defined in `waitset_chan.h`.

Once a channel is in the *pending* state, if it is not deregistered it will eventually be *delivered* to a user of the waitset (calling a function such as `get_next_event()` or `event_dispatch()`) unless it is deregistered beforehand. Once the event is delivered a channel reverts to *unregistered* state. Event registrations are therefore single-shot; an event handler function that wishes to handle future events will typically re-register itself.

4.1.2 Waitset API

Waitset creation/destruction

```
void waitset_init(struct waitset *ws);
errval_t waitset_destroy(struct waitset *ws);
```

These functions initialise the state of a new waitset, or destroy the state of an existing waitset.

Event dispatch

```
errval_t get_next_event(struct waitset *ws, struct event_closure *retclosure);
```

This function is the core of the event handling mechanism. It returns the next pending event on the given waitset, if necessary by blocking the calling thread until an event is triggered.

```
errval_t event_dispatch(struct waitset *ws);
```

This function is a wrapper for `get_next_event()` which also invokes the event (by calling `event.handler(event.arg)`) before returning. It is typically used within an infinite loop on an event-handling thread.

4.1.3 Implementation

Internally, the waitset maintains a queue of waiting threads, and three queues of channels: one each for channels in the *idle*, *pending* and *polled* state. The majority of the implementation consists of manipulating these queues, and implementing the state transitions described in the previous subsection.

The core logic of the waitset lies in the `get_next_event()` function, whose operation is as follows:

1. If there are any channels in the pending queue, remove the next channel from the queue, mark it unregistered, and return its event closure.
2. Otherwise, if there are any channels in the polled queue, and no other thread is already polling channels on this waitset, poll channels in the polled queue long for as long as the pending queue remains empty. The polling mechanism is presently used only by the UMP interconnect driver, and is described in more detail in Section 5.2.
3. Otherwise, block on the queue of waiting threads. When unblocked (either when a channel becomes pending or when needed to begin polling), resume from step 1.

4.2 Interconnect drivers

Interconnect drivers implement the runtime support needed by specific message transport mechanisms. Chapter 5 describes the specifics of each interconnect driver. Although there is no uniform interface to the interconnect drivers, they typically support the following:

- Binding (connection setup), as described in Section 4.3
- A mechanism to *send* a message on a channel
- A mechanism to *receive* messages from a channel
- An interface to the waitset, allowing the user to *register* for event notifications when it is possible to send or receive on a channel
- Various channel-specific control operations

In general, messages at this level of the system are word-granularity fixed-size (or variable-size with an upper bound) untyped data.

The interconnect drivers that are available in a given domain are determined by the compile-time configuration and processor architecture. The only driver that is always present is local message passing (LMP), which provides kernel-mediated communication with other dispatchers on the same core, and is used to communicate with the local Monitor.

4.3 Export and binding

The export and binding process for all interconnect drivers is mediated by the monitors. Section 3.1 documents the user-level API provided by the binding objects, while here we describe what happens under the covers.

The monitors are responsible for allocating interface references, or irefs, tracking the association of irefs to the dispatcher implementing that service, and mediating the bind process when a client wishes to bind to an iref. In order to boot-strap this mechanism, every dispatcher is always created with a binding to its local monitor, as described in Section 5.1.1.

4.3.1 Export

In order to export a new service, a dispatcher sends an `alloc_iref_request` message to its local monitor, passing an opaque service identifier (which is unique to the dispatcher). The monitor constructs an iref, and returns it to the user along with the service identifier in an `alloc_iref_reply` message.

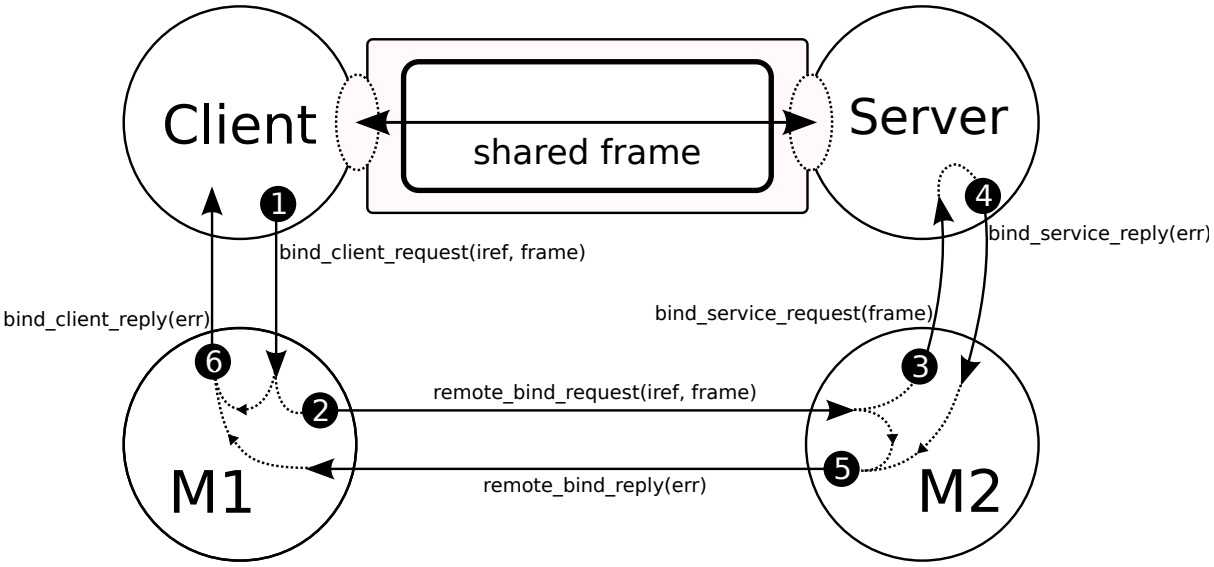


Figure 4.1: UMP bind process

4.3.2 Binding

In order to bind, a client dispatcher on a core presents the iref to its local monitor, and requests to bind with a specific interconnect driver. Although different interconnect drivers use different messages and parameters, the process is very similar for all, so by way of example we present here the binding mechanism for the UMP interconnect driver.

Figure 4.1 shows the steps in setting up UMP binding:

1. The client sends `bind_ump_client_request` to its local monitor, with the iref, the client's (opaque) connection ID, and various parameters that are specific to the interconnect driver. For UMP, this includes a capability to a shared frame, which will be used for message transfer.

2. The local monitor determines (from the iref) the core on which the service resides, and forwards the request to the monitor on that core using a `bind_ump_request` message (NB: mislabelled `remote_bind_request` in the figure).

3. The monitor on the server's core determines the local dispatcher which has exported this iref, and forwards the bind request to it using a `bind_ump_service_request` message. It also provides the opaque service identifier which the dispatcher gave when originally exporting the service.

4. The server dispatcher uses the service identifier to locate and invoke the connection callback for the given service. Assuming this succeeds, it allocates its local state for the binding (e.g. the binding object). It then replies to its local monitor with a `bind_ump_reply_monitor` message (labelled `bind_service_reply` in the figure).

5. The monitor proxies the reply back to the monitor on the client's core

6. The monitor on the client core proxies the reply back to the client, providing the client's connection ID, which allows it to locate the binding object and run the bind callback. The binding has now completed.

The binding mechanisms for other interconnect drivers (e.g. LMP, BMP) are similar, in that they involve an exchange of capabilities between client and server mediated by the monitors. Once the binding is established, direct communication can take place using the exchanged resources.

4.4 Indirect capability transfer

Because capabilities are maintained as references to per-core state in the CPU drivers, only the LMP interconnect driver, which traverses kernel-mode code, can directly deliver a capability along with untyped message payload. For other interconnect drivers, such as UMP, capabilities travel out-of-band from message payload through the monitors.

The process of indirect transmission of capabilities is as follows:

1. The binding object waits to acquire a mutex on its local monitor binding. This is necessary, because it must send a message to its local monitor, and the local monitor binding is shared by all bindings (and binding-related code) executing on the same dispatcher. While waiting, it sends the non-capability payload of the message as usual.
2. Upon receipt of the first message payload, the receiver sends back an acknowledgement that it is ready to receive a capability. This handshake is necessary to avoid over-running the receiver's buffers, and to ensure that capabilities can be associated with the correct messages.
3. Once the monitor binding mutex is acquired, and the capability acknowledgement has been seen, a `cap_send_request` message is sent to the local monitor, containing the capability and an identifier for the channel on which the capability is to be sent.
4. The monitor determines whether the cap may be sent to the remote core. A full discussion of the rules used is beyond the scope of this document, but in general capability types that refer to inherently local state (such as LMP endpoints) may not be sent, nor may capabilities that are currently being revoked. If the check fails, a `cap_send_reply` error is sent back to the local dispatcher; if it succeeds, the capability is serialised and proxied to the remote monitor as another `cap_send_request` message on the inter-monitor channel.
5. The remote monitor reconstructs the capability from its serialised representation, and forwards it on to the destination dispatcher with a `cap_receive_request` message.
6. The receiver identifies the binding to which the capability belongs, and invokes a callback on that binding. This in turn stores the capability in the appropriate location.
7. When the last capability and message payload are received, the binding object delivers the message to the user as usual.

Chapter 5

Interconnect drivers

5.1 LMP: local (intra-core) message passing

IDC between dispatchers on the same core uses LMP (local message passing). LMP uses endpoint capabilities for communication. In this case the channel struct contains a reference to a remote endpoint cap (for the sender) and a local endpoint cap (for the receiver). It also contains a reference to an endpoint structure. This is a struct contained in the receiver's dispatcher and refers to a buffer that is used to store incoming messages. The dispatcher can have multiple such endpoint buffers.

The sender's remote endpoint cap is derived from the receiver's dispatcher cap, and contains a pointer back to the DCB and an offset into the associated endpoint buffer.

When a message is sent, the sender invokes the remote endpoint cap with the message to send. This causes the kernel to copy the message into the associated endpoint buffer and then make the receiver dispatcher runnable (which will cause it's `run()` upcall to be invoked when it is scheduled). The run function then finds the endpoint with a message in it and invokes the trigger on the appropriate waitset. The channel's event handler is responsible for getting the rest of the message out of the buffer and invoking appropriate user-defined handlers.

Sender: `lmp_deliver_payload (dispatch.c)`

Receiver: `disp_run (dispatch.c)` `lmp_endpoints_poll_disabled (lmp_endpoints.c)`

Event handler: `myappif_lmp_rx_handler (myappif_flounder_bindings.c)`

Binding: The process of binding for LMP involves the sender creating the endpoint capability and

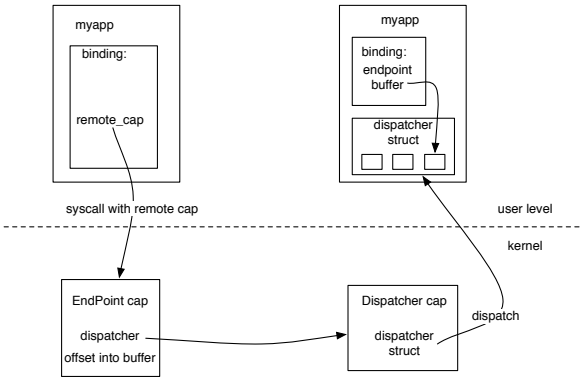


Figure 5.1: Local Message Passing

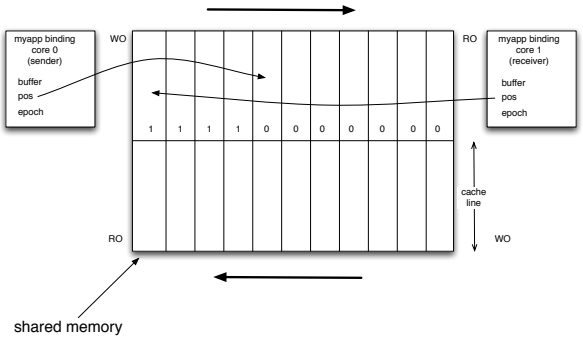


Figure 5.2: User-level Message Passing

appropriate buffers, then invoking `lmp_bind_request` on the monitor. The monitor sends the request to the receiver, where it sets up its own endpoint cap and buffer and binding struct, and returns the cap through the monitor to the sender.

Passing capabilities - LMP: When a capability is sent in a message, then the kernel must additionally copy the cap to the receiver's cspace (a slot for this is setup in the endpoint). The receive handler must ensure that it copies the cap out of this cspace slot in order to free it for future transfers.

5.1.1 Messaging with the local Monitor

5.2 UMP: user-level message passing with coherent shared memory

IDC between dispatchers on different cores uses UMP (user-level message passing) (on x86).

UMP uses shared memory and polling and inter-processor interrupts for communication. The shared memory is split into a send buffer and a receive buffer (Figure 5.2), with each entry in the buffers being the size of a cache line. The sender writes into the send buffer, and the receiver reads from it. The receiver polls to determine whether new messages have been received.

The binding struct for a UMP channel contains UMP specific channel structs (nested within various included structs) which contain a reference to the shared memory, as well as a counter specifying the current read or write position.

In order to detect whether the write position has been reached when reading, the channel also contains an epoch flag. Every buffer entry has an epoch bit that is flipped every time a new entry is added. When reading a new entry the channels's epoch is compared to the epoch bit of the buffer slot to be read. If it is the same, then the entry is valid, if it is different then it is not valid. The channel's epoch flag is flipped every time the counter wraps around.

The sender also has to know whether it is going to run up against the receivers's read position. Rather than flip epoch bits on a read (in order to avoid bouncing cache lines the buffer is kept write-only for the sender and read-only for the receiver) each message has a sequence id. When replying to the sender the receiver sends its latest read sequence id. The sender then knows how many messages are outstanding and together with knowledge of the buffer size, can avoid overwriting messages that haven't been read yet.

Binding: The UMP binding process involves the source creating the channel buffers, then sending a cap to the buffer along with a bind request through LMP to the monitor. The monitor serialises the cap and sends it through to the destination monitor. The destination monitor recreates the cap, and sends it on to the receiver. This sets up its channel struct accordingly and finalises the binding.

Passing capabilities - UMP: UMP messages containing capabilities must be sent in two phases. The non-cap message is sent as normal. The capability must be sent through the monitor as only the monitor is able to invoke a syscall to serialise and deserialise a capability. The sender sends the capability with a request to the monitor, the monitor serialises it and sends it to the destination monitor, where it is deserialised, and send through LMP to the destination.

5.3 RCK: message passing on the SCC

5.4 BMP: Beehive message passing

Chapter 6

Additional Flounder backends

6.1 Loopback

6.2 Message buffers

6.3 RPC Client

6.4 THC

6.5 AHCI Command Translator

This backend is documented in the AHCI TN.

Chapter 7

Name service

TODO

Chapter 8

Interaction with Hake build system

TODO

- Configuration options
- Contents of Hakefiles
- Location/contents of generated files
- Dependencies

Chapter 9

Unsupported features and future work

TODO

- Connection/binding teardown
- Variable-length arrays of types other than `char`, `int8` and `uint8`
- Types shared by multiple interfaces, control over C naming of generated type definitions
- Non-blocking variant of `get_next_event()`
- Fast LRPC optimisations (combines waitset, thread switch and send)

References

[1] Barrelfish Project. Mackerel 1.2 User Guide. Barrelfish Technical Note 002, Systems Group, ETH Zurich, April 2010.