Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Inter-dispatcher communication

Andrew Baumann

Introduction

- Tutorial on inter-dispatcher communication (IDC)
  - May be between dispatchers within the same domain
- Abstraction of typed bidirectional message-passing
- Event driven, asynchronous and non-blocking

Interface definition language

Defines message types and arguments on a given connection

```
interface test "Test interface" {
    /* Type definitions and aliases */
    alias number uint32;
    typedef enum {FOO, BAR, BAZ} myenum;

    /* Message definitions */
    message basic(number arg, myenum e);
    message str(number arg, string s);
    message caps(number arg, cap cap1, cap cap2);
    message buffy(uint8 buf[buflen]);
};
```

Interface `test` defined in `barrelfish/if/test.if`

Flounder stub compiler

- Generates stubs to send and receive messages
  - Marshalling, demarshalling, fragmentation, etc.
- Implementation varies: different interconnect drivers
- Before dispatchers can communicate, they must bind
- Binding process differs between “server” and “client”
  - Equivalent once bound

Binding: server-side

Export a new instance of the `test` service:

```
#include <if/test_defs.h>

struct waitset *waitset = get_default_waitset();
errval_t err = test_export(NULL /* user state */, export_callback,
                           connect_callback, waitset,
                           IDC_EXPORT_FLAGS_DEFAULT);
if (err_is_fail(err)) {
    ...
}
```

- Waitset is part of event handling mechanism (more later)
- Success indicates that export process has begun
- `export_callback()` invoked when export completes
- `connect_callback()` invoked when client attempts to connect

Binding: server-side

Export callback

```
static void export_callback(void *st, errval_t err, iref_t myiref)
{
    if (err_is_fail(err)) {
        ... // report error
    } else {
        printf("service exported at iref %"PRIuIREF"\n", myiref);
        ... // now what?
    }
}
```

- Invoked when export process completes
- iref: Interface reference for this instance of the service
- Required by potential clients, must communicate to them
- e.g. via name service

Name service

Stores global mapping of names (arbitrary strings) to irefs

Server code (e.g. in export callback):

```
#include <barrelfish/nameservice_client.h>

err = nameservice_register("test service", myiref);
assert(err_is_ok(err));
```

Binding: client-side

1. Name service lookup:

```
iref_t srviref;
err = nameservice_blocking_lookup("test service", &srviref);
if (err_is_fail(err)) {
    ...
}
```

   - Call blocks until name is registered (possibly forever)
   - Used to resolve races between client and server startup
   - Non-blocking version also available

2. Initiate binding process:

```
err = test_bind(srviref, bind_callback, NULL /* state pointer */,
                get_default_waitset(), IDC_BIND_FLAGS_DEFAULT);
if (err_is_fail(err)) {
    ...
}
```

   - `bind_callback()` invoked when bind completes

Binding: server-side

Connect callback

```
static errval_t connect_callback(void *st, struct test_binding *b)
{
    printf("service got a connection!\n");

    // TODO: setup binding object

    return SYS_ERR_OK; // accept the connection
}
```

- Binding object provided (`struct test_binding`)

Binding: client-side

Bind callback

```
static void bind_callback(void *st, errval_t err,
                          struct test_binding *b)
{
    if (err_is_fail(err)) {
        ... // bind failed
    } else {
        printf("client bound!\n");
        // TODO: setup binding object
    }
}
```

- Each side now has a binding object for the other

Binding objects

- Handle to communication endpoint
- Virtual function API

```
struct test_binding {
    void *st;                    // user's state pointer
    struct waitset *waitset;     // (read only to user)
    struct event_mutex mutex;    // for multithreaded apps

    struct test_tx_vtbl tx_vtbl;          // provided by binding
    struct test_rx_vtbl rx_vtbl;          // filled in by user
    test_error_handler_fn *error_handler; // filled in by user

    /* control functions, provided by binding */
    test_can_send_fn *can_send;
    test_register_send_fn *register_send;
    test_change_waitset_fn *change_waitset;
    test_control_fn *control;
};
```

Receiving messages

1. Define message handlers and vtable

```
static void rx_basic(struct test_binding *b, uint32_t arg,
                     test_myenum_t e)
{
    printf("got a basic message %u %d\n", arg, e);
}

static void rx_str(struct test_binding *b, uint32_t arg, char *s)
{
    printf("got a string message %u '%s'\n", arg, s);
    free(s); // reference args allocated on heap, property of handler
}

static struct test_rx_vtbl my_rx_vtbl = {
    .basic = rx_basic,
    .str = rx_str,
    // ... other handlers
};
```

Receiving messages

2. Copy message receive handler vtable to the binding

   In `bind_callback()` and `connect_callback()`:

```
b->rx_vtbl = my_rx_vtbl;
```

3. Process events on the waitset

```
#include <barrelfish/waitset.h>

struct waitset *ws = get_default_waitset();
while (true) {
    errval_t err = event_dispatch(ws);
    if (err_is_fail(err)) {
        USER_PANIC_ERR(err, "fatal error in event loop");
    }
}
```

Sending messages

`<if/test_defs.h>` provides:

```
errval_t test_str__tx(struct test_binding *_binding,
                      struct event_closure _continuation,
                      uint32_t arg, const char *s)
{
    return _binding->tx_vtbl.str(_binding, _continuation, arg, s);
}
```

So, to send a string:

```
errval_t err = test_str__tx(binding, NOP_CONT, 1234, "hello world");
if (err_is_fail(err)) {
    ... // handle errors
}
```

Sending messages

Buffering guarantees

- Success return indicates only that a message was queued
  - May take arbitrary time to transmit on underlying channel
- Reference params (strings, arrays, caps) must remain live
- Send continuation invoked when message has been sent
  - `NOP_CONT`: no continuation
  - `MKCONT(func,arg)`: constructs continuation closure
  - e.g. `free(str)` after string message is sent:

```
err = test_str__tx(binding, MKCONT(free, str), 1234, str);
```

- Queue limited to one outgoing message
- Subsequent sends fail with `FLOUNDER_ERR_TX_BUSY`

Sending messages

Handling TX_BUSY errors

- Serialise sends with continuation function
  - See `usr/tests/idctest/idctest.c` for an example
- Register for a callback when we can send again:

```
if (err_no(err) == FLOUNDER_ERR_TX_BUSY) {
    struct waitset *ws = get_default_waitset();
    err = binding->register_send(binding, ws,
                                 MKCONT(retry_send,binding));
    if (err_is_fail(err)) {
        ... // why might this fail?
    }
}
```

  - Only one callback may be registered at a time
  - In practice, may need to implement arbitrary queue
  - See `usr/ramfsd/service.c` for an example
- Use THC!

Handling asynchronous errors

- Connection teardown, fatal errors, missing capabilities, etc.
- Binding reports errors via `error_handler` callback

```
static void my_error_handler(struct test_binding *b, errval_t err)
{
    // log error, disconnect client, destroy binding, etc.
}
```

Setup in `bind_callback()` and `connect_callback()` with:

```
b->error_handler = my_error_handler;
```

Concurrency on bindings

- Binding objects are not thread-safe
  - Motivation: pure event-driven applications
- Must protect against concurrent access to same binding
- `mutex` field provided for this purpose:

```
event_mutex_threaded_lock(&binding->mutex);
...
event_mutex_unlock(&binding->mutex);
```

- Could use other mechanisms

RPC client stubs

- No pipelining: single request outstanding on a channel
- Client-side only
- Synchronous, blocking, slow
- No event handlers or `TX_BUSY` errors to deal with

1. Define `rpc` messages in interface:

```
interface test2 {
    rpc testrpc(in string s, out uint32 result);
}
```

This is basically equivalent to:

```
interface test2 {
    message testrpc_call(string s);
    message testrpc_response(uint32 result);
}
```

Server must send corresponding `response` message for every `call`

RPC client stubs

2. Bind as usual
3. Construct RPC client object above binding:

```
#include <if/test2_rpcclient_defs.h>

struct test2_binding *b = ... // normal bind process
struct test2_rpc_client *cl = malloc(sizeof(struct test2_rpc_client));
errval_t err = test2_rpc_client_init(cl, b);
if (err_is_fail(err)) {
    ...
}
```

4. Make RPC calls:

```
uint32_t result;
err = cl->vtbl.testrpc(cl, "input string", &result);
if (err_is_fail(err)) {
    ...
}
```

Bulk data transfer

- Library for transferring untyped bulk data via (possibly non-coherent) shared memory
- Complements IDC mechanisms
- API not yet documented/finalised
- See `<barrelfish/bulk_transfer.h>` in the meantime

Behind the scenes

- Multiple implementations, one per interconnect driver

LMP	Local (same-core) message passing, mediated by CPU driver
UMP	Coherent shared memory transport (on x86)
RCK	Non-coherent transport for Intel SCC
BMP	Beehive hardware messaging
Loopback	Fakes out “IDC” on same dispatcher

- Binding process chooses appropriate interconnect driver
- Marshalling, demarshalling, fragmentation, flow control
- Capabilities may travel out-of-band via monitors

Features not yet implemented

- Connection teardown
- Variable-length arrays (other than byte arrays)
- Type definitions shared between multiple interfaces
- Control over C naming of generated type definitions

Summary

- Flounder IDC provides point-to-point typed messages
- Asynchronous, with explicit event-driven API
- Lowest common API of various interconnect drivers
- Basis for all communication in Barrelfish
- Extended by THC and routing/group communication

Further information:

- Barrelfish Technical Note 011: IDC
- `usr/tests/idctest/idctest.c`