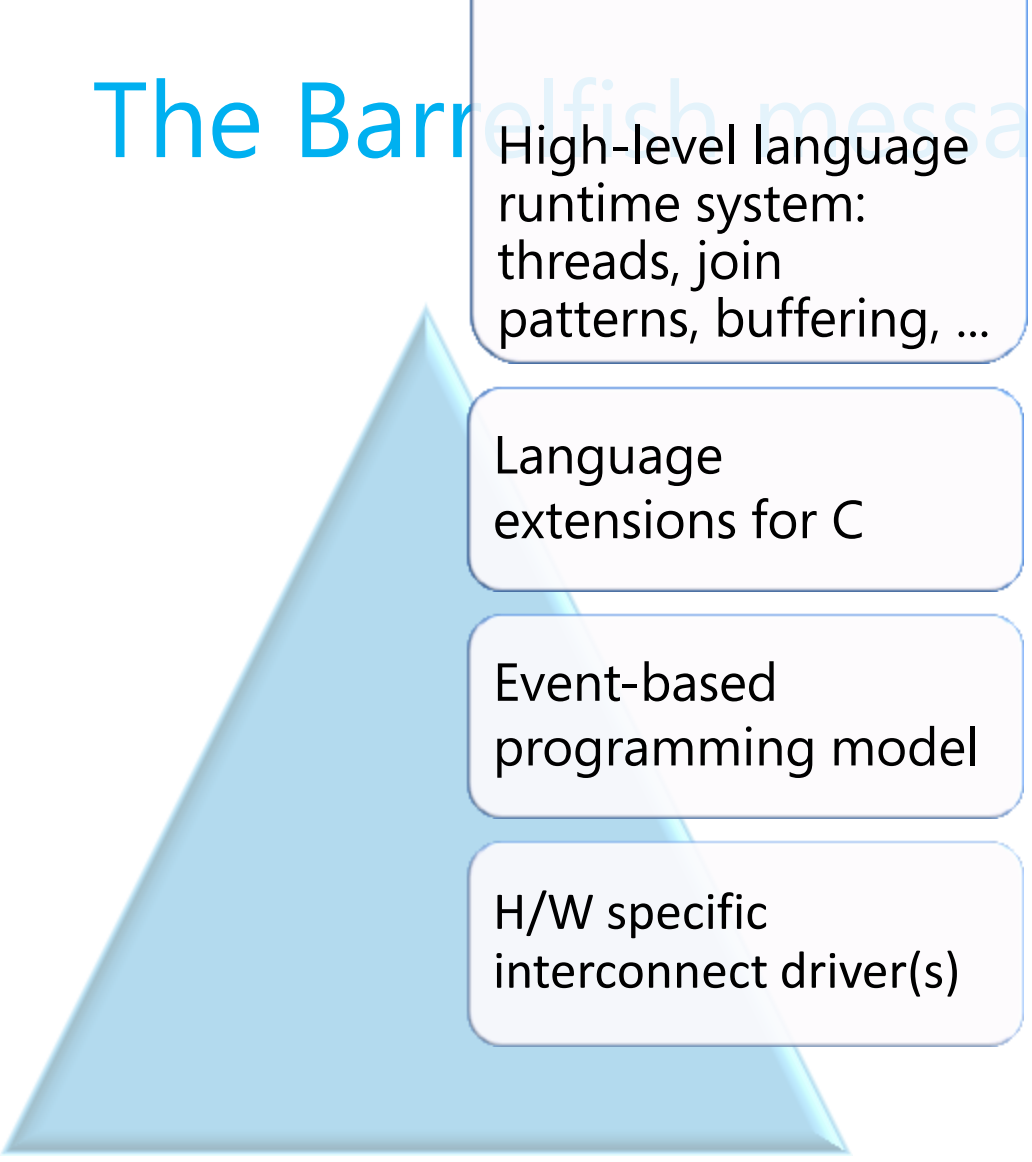
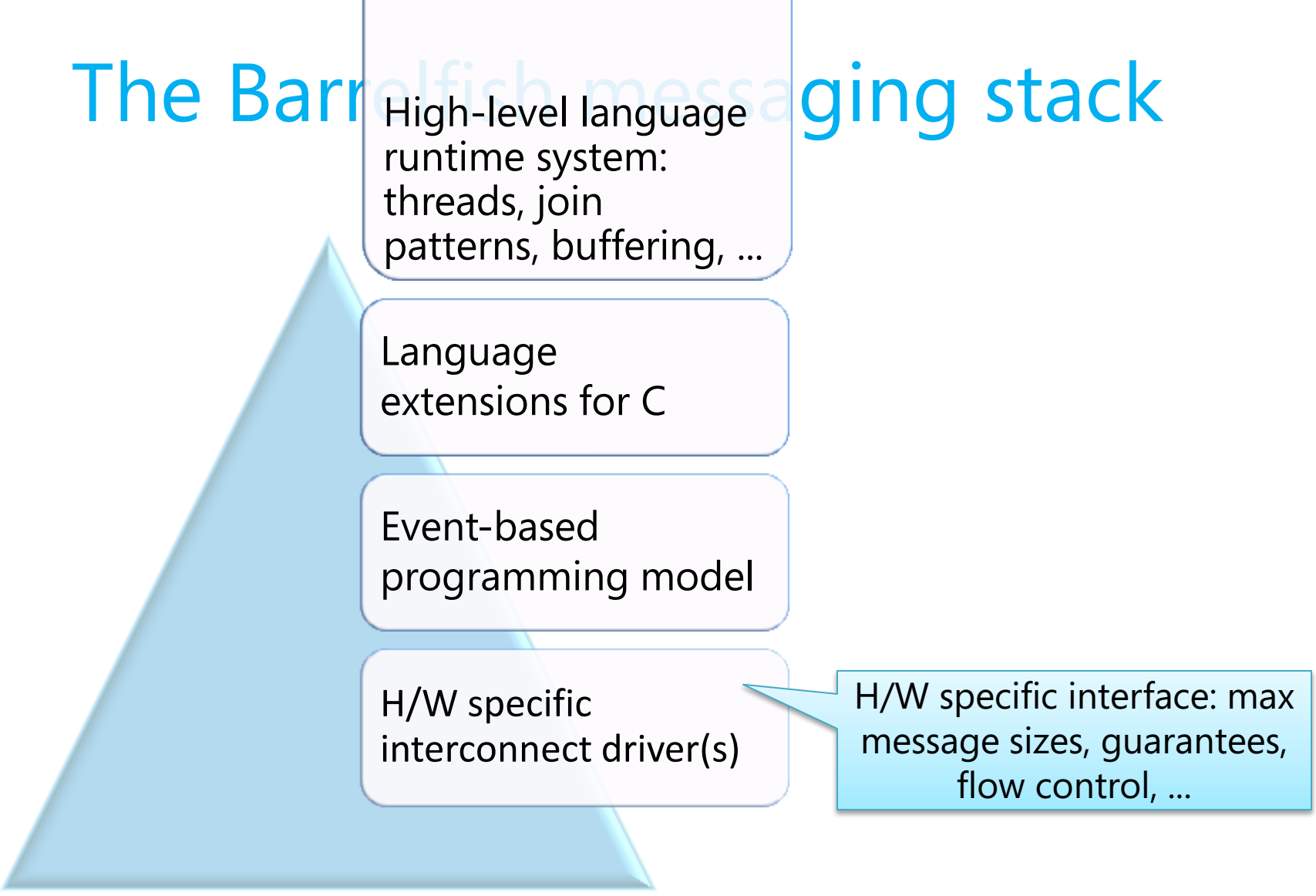
Barrelfish language extensions

Tim Harris

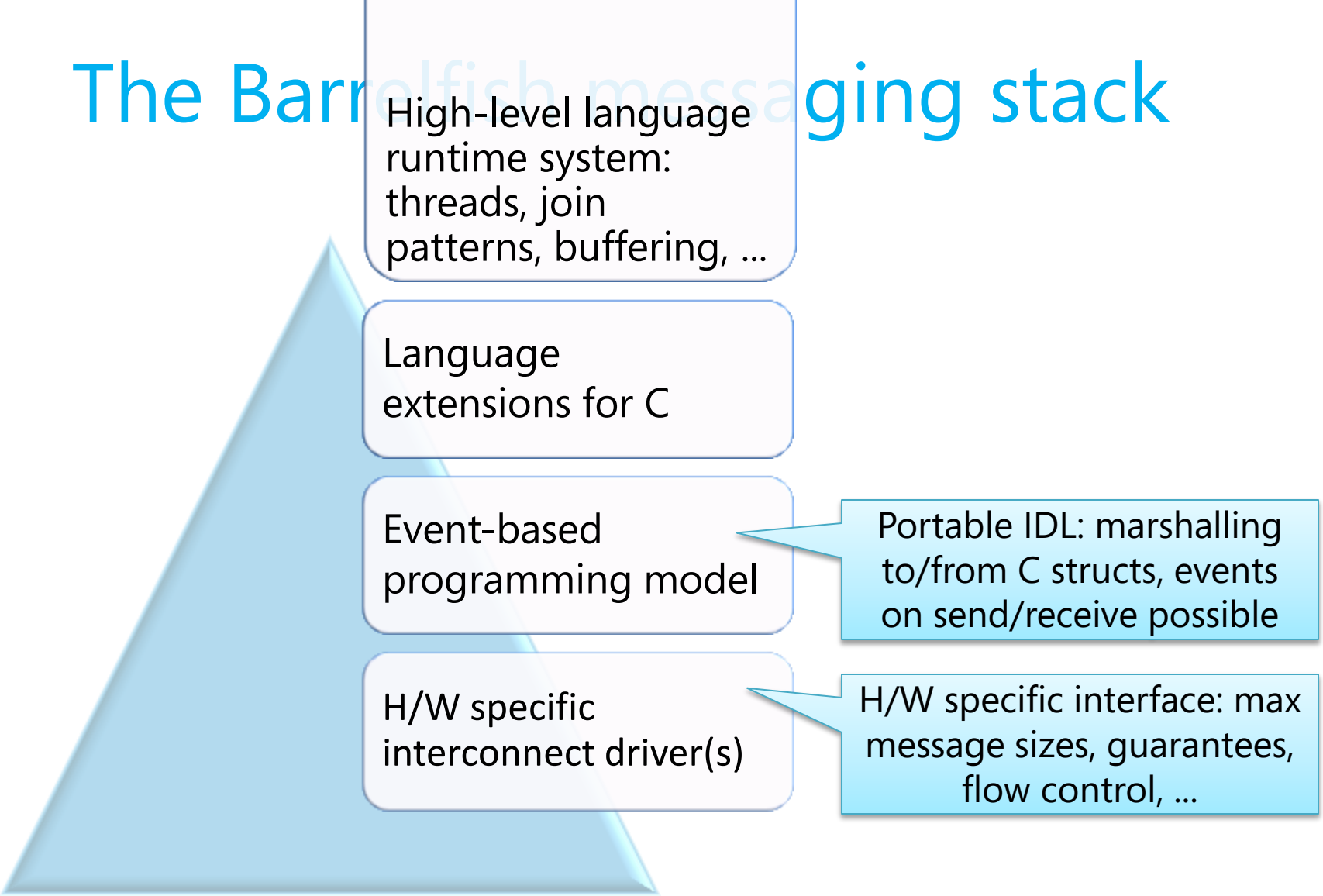
The Barrelfish messaging stack



The Barrelfish messaging stack



The Barrelfish messaging stack



The Barrelfish messaging stack

High-level language runtime system: threads, join patterns, buffering, ...

Language extensions for C

Simple synchronous send/receive interface, support for concurrency

Event-based programming model

Portable IDL: marshalling to/from C structs, events on send/receive possible

H/W specific interconnect driver(s)

H/W specific interface: max message sizes, guarantees, flow control, ...

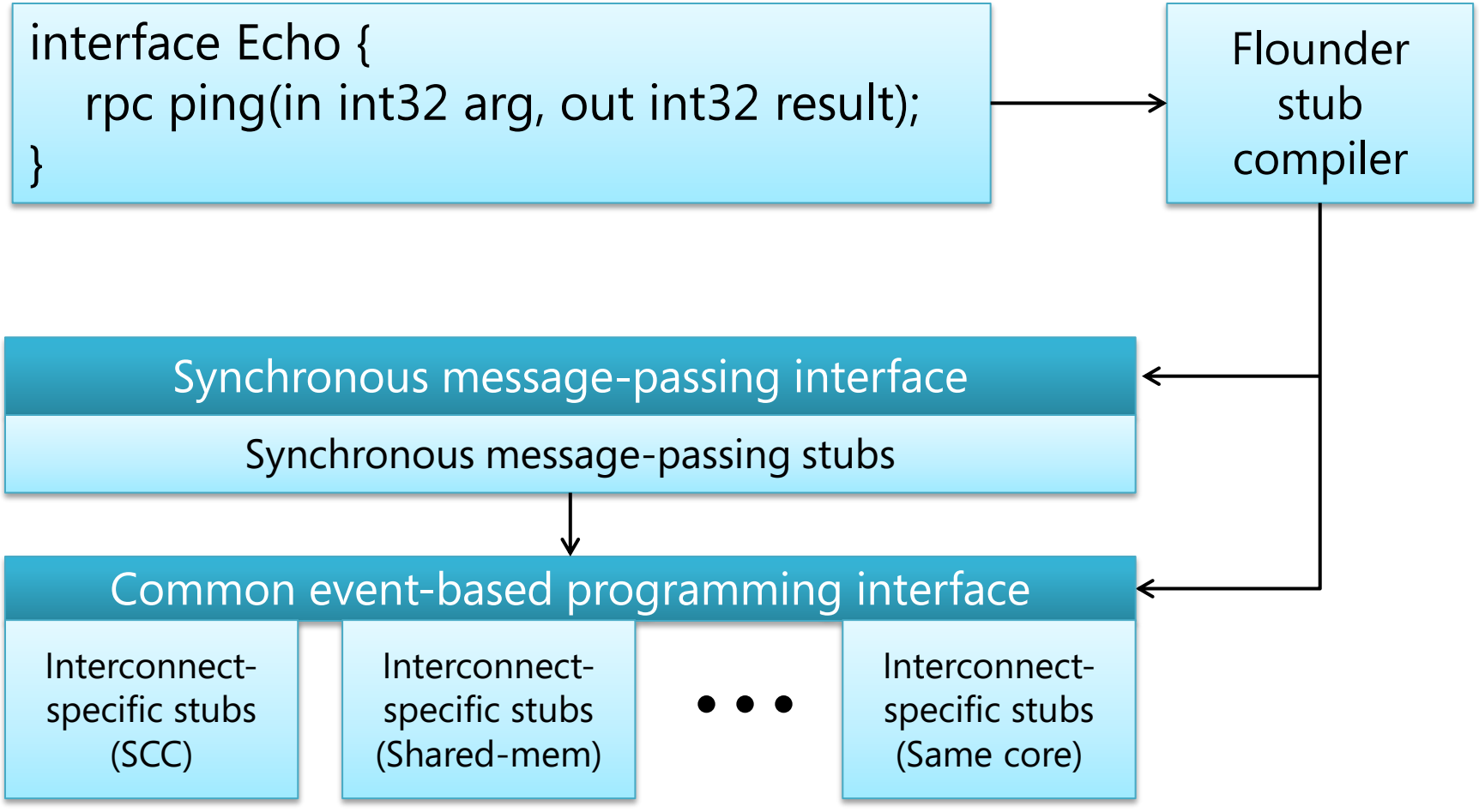
Synchronous send/receive

Supporting concurrency

Cancellation

Performance

Synchronous message-passing



Synchronous message-passing

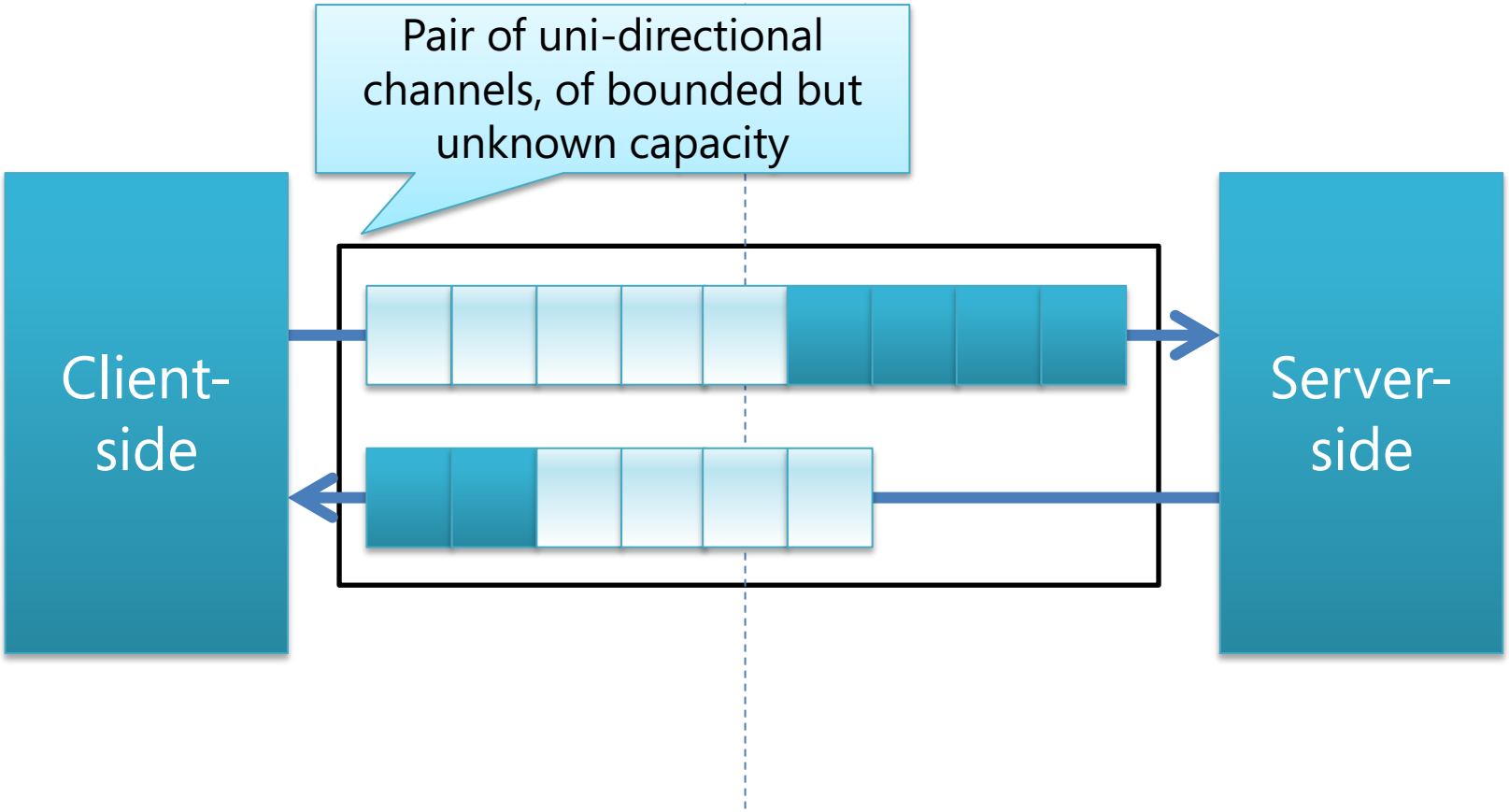
```
interface Echo {
    rpc ping(in int32 arg, out int32 result);
}
```

Flounder stub compiler

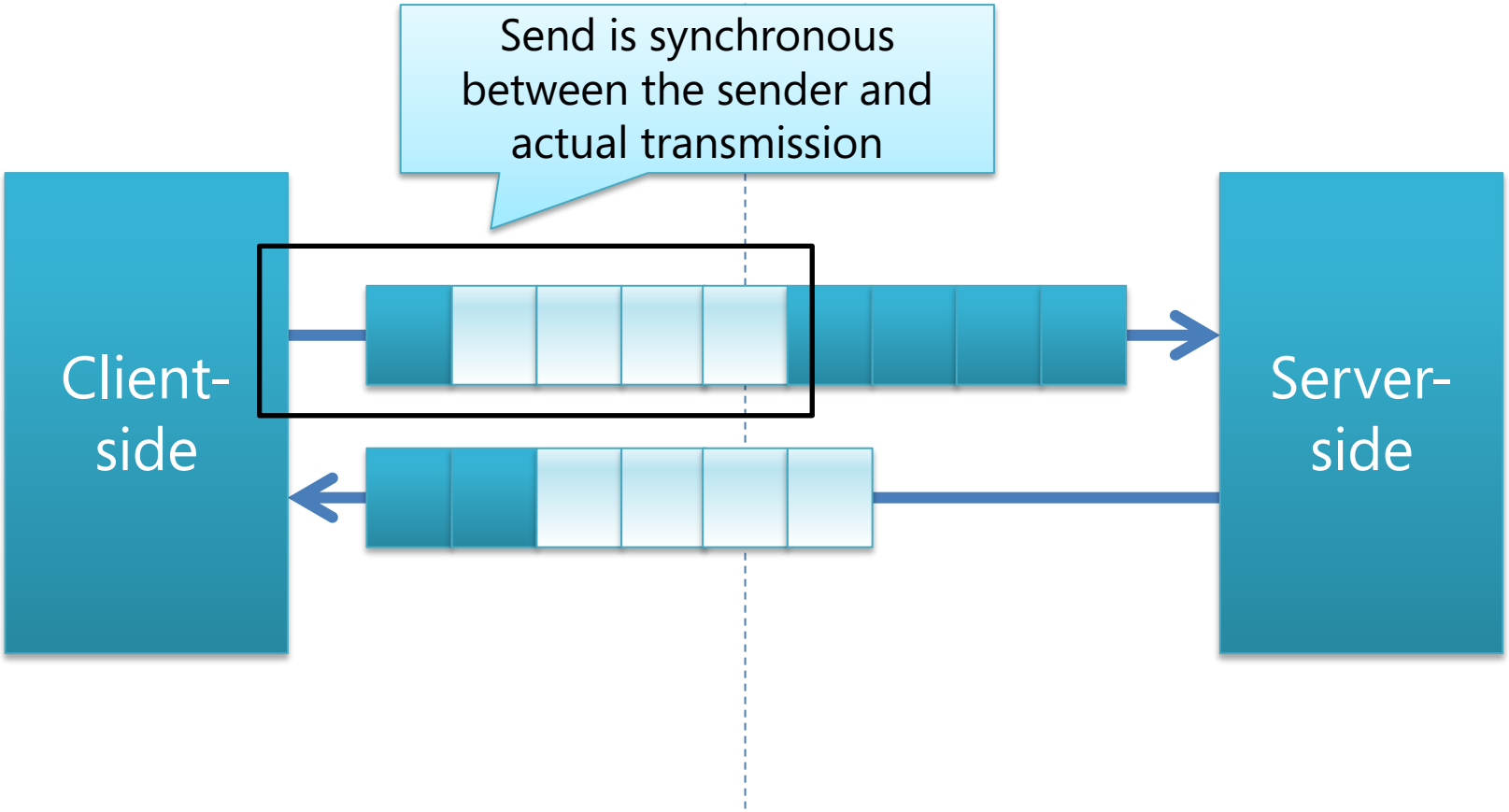
```
// Send "ping", block until complete
void Echo_tx_ping (Echo_binding *b, int arg);

// Wait for and receive response to "ping"
void Echo_rx_ping (Echo_binding *b, int *result);

// RPC send-receive pair
void Echo_ping (Echo_binding *b, int arg, int *result);
```

(SCC) (UMP) (LMP)

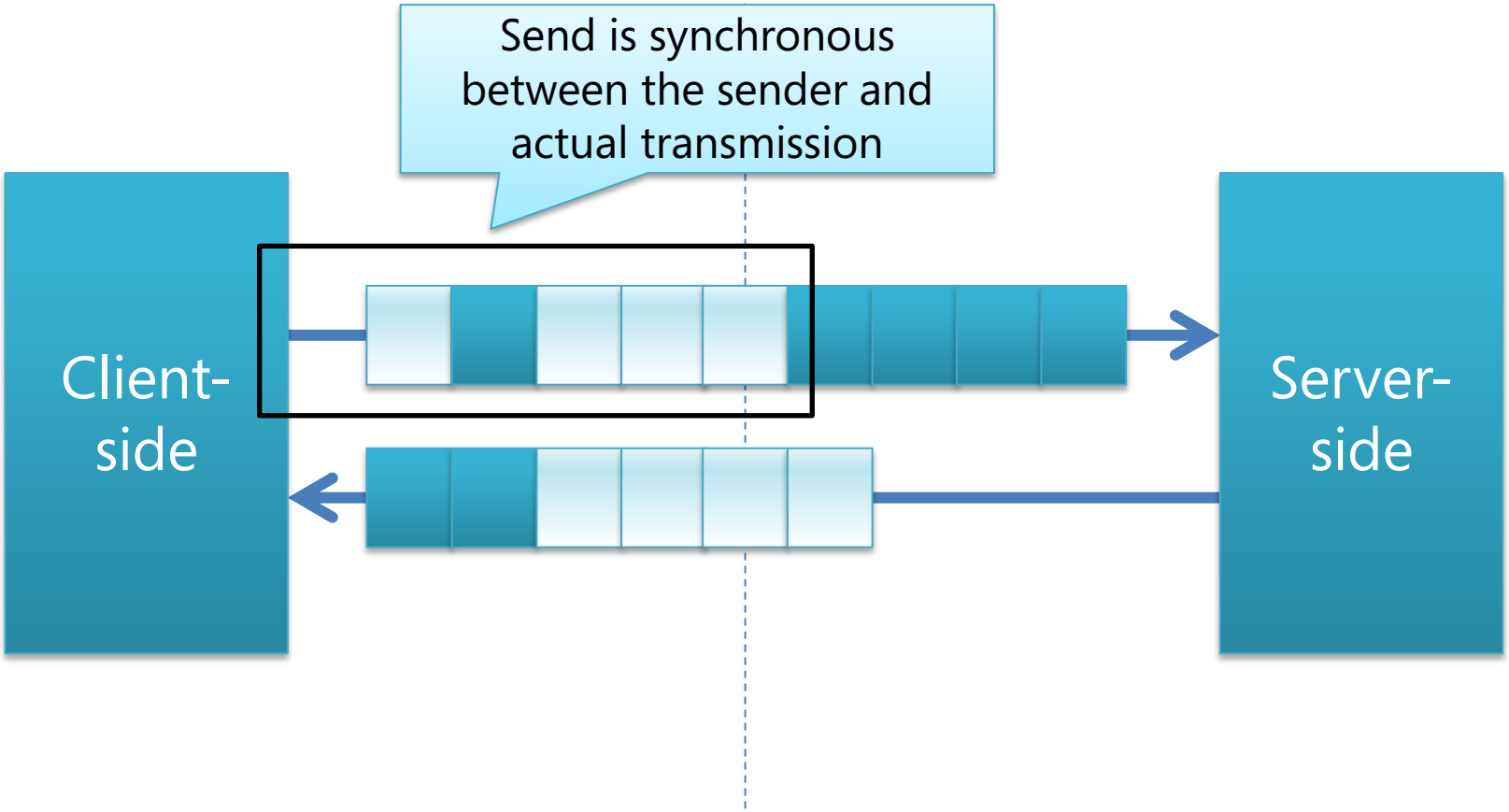
Channel abstraction



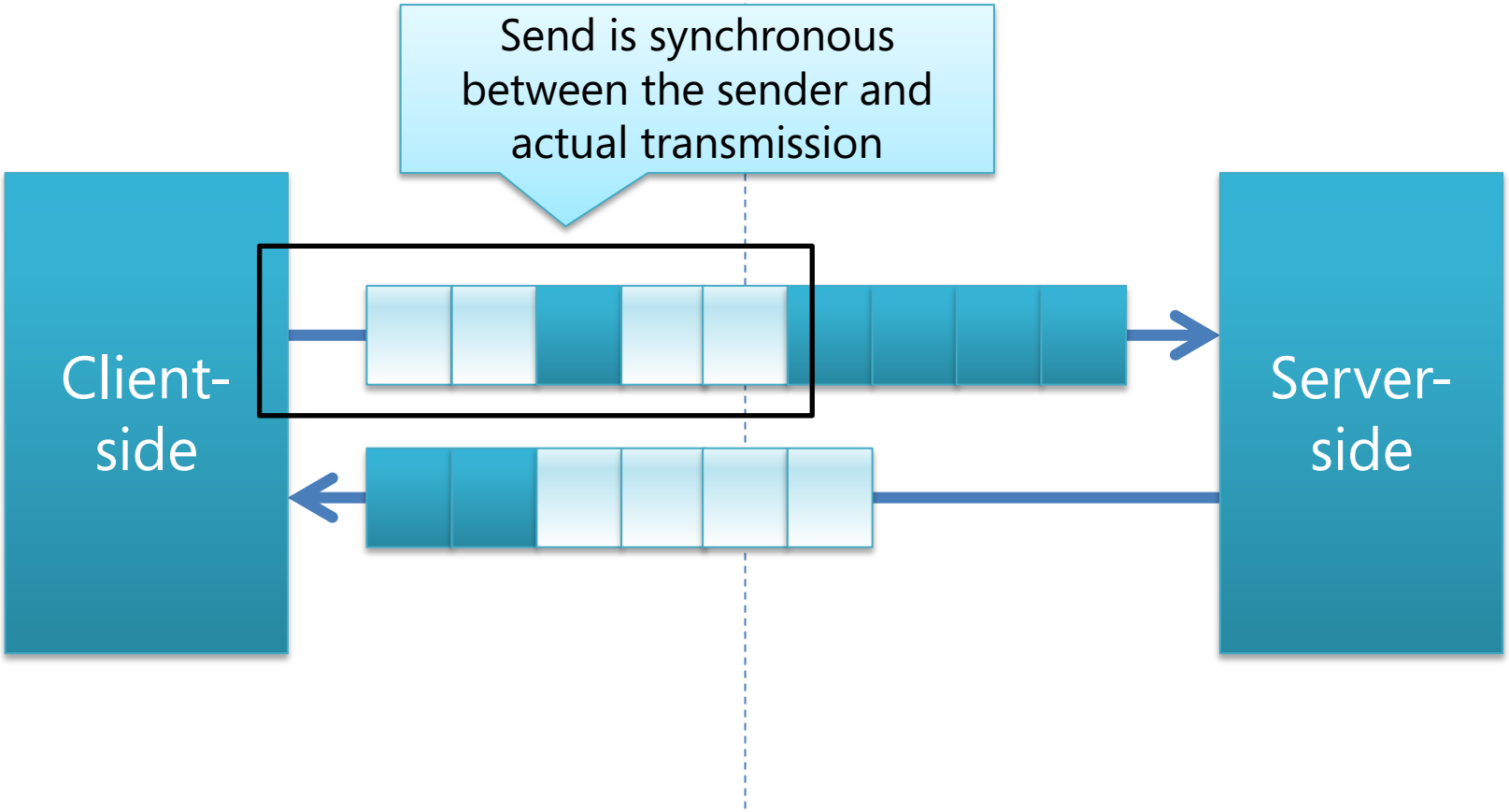
Channel abstraction



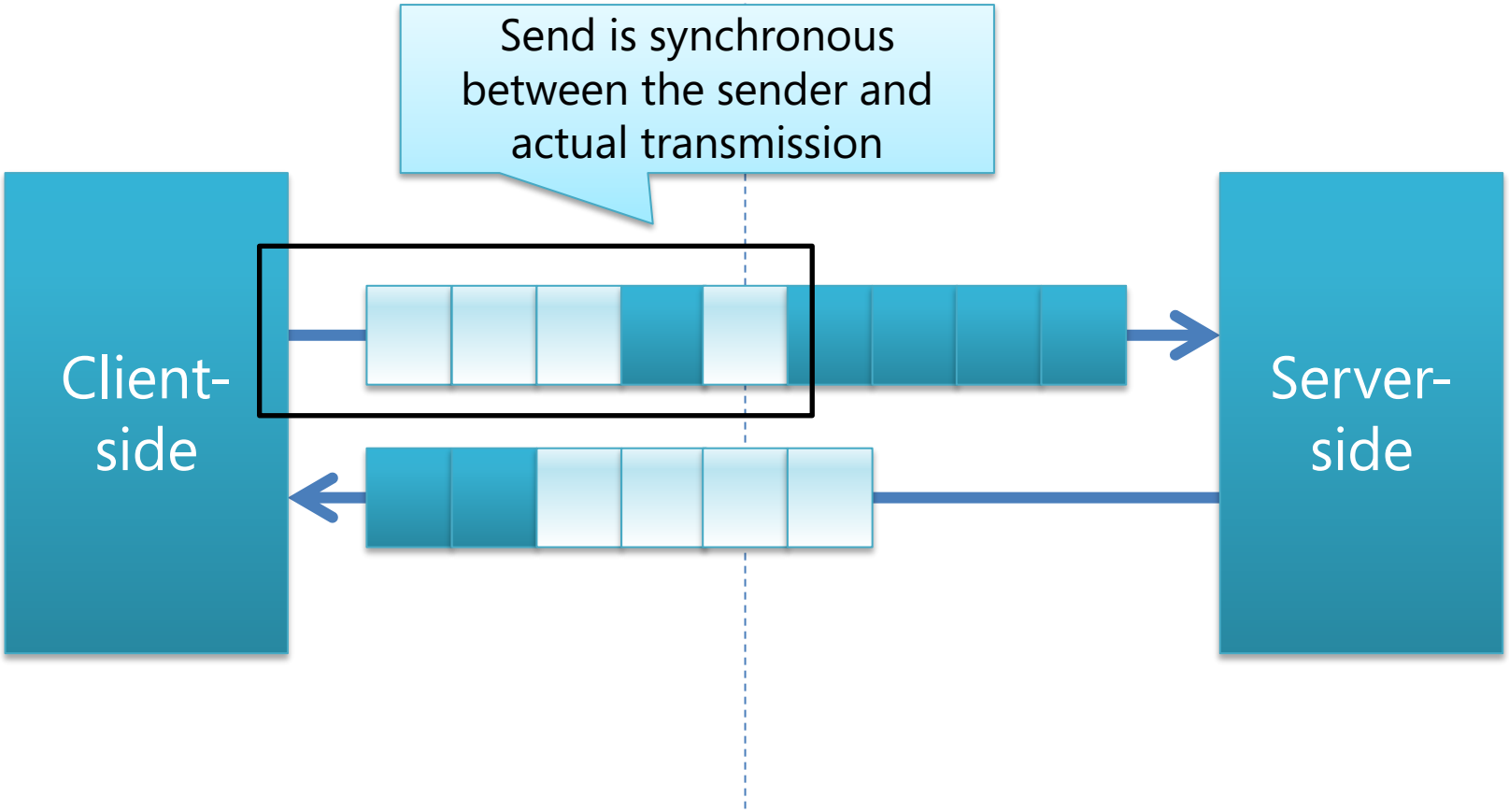
Channel abstraction



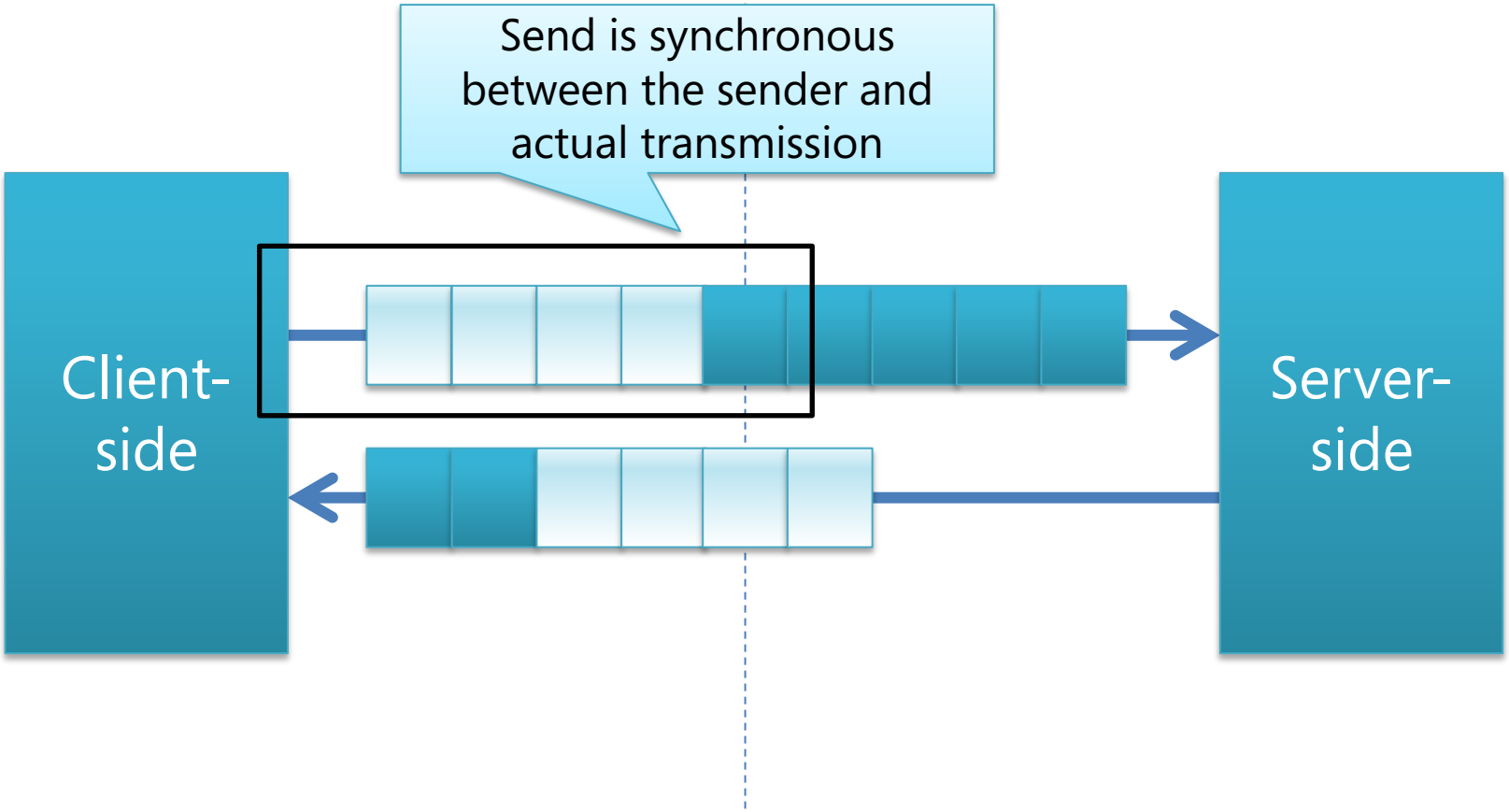
Channel abstraction



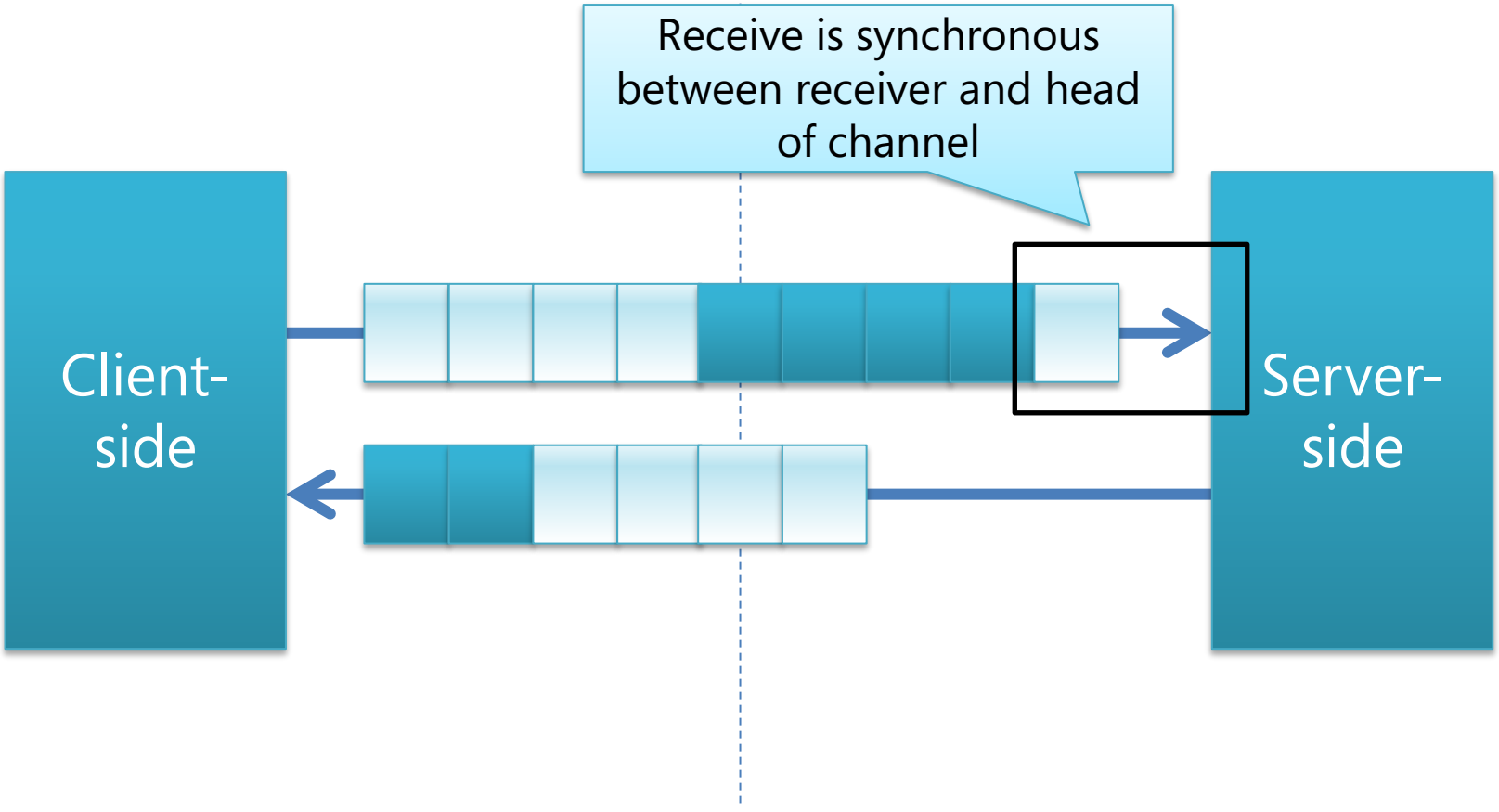
Channel abstraction



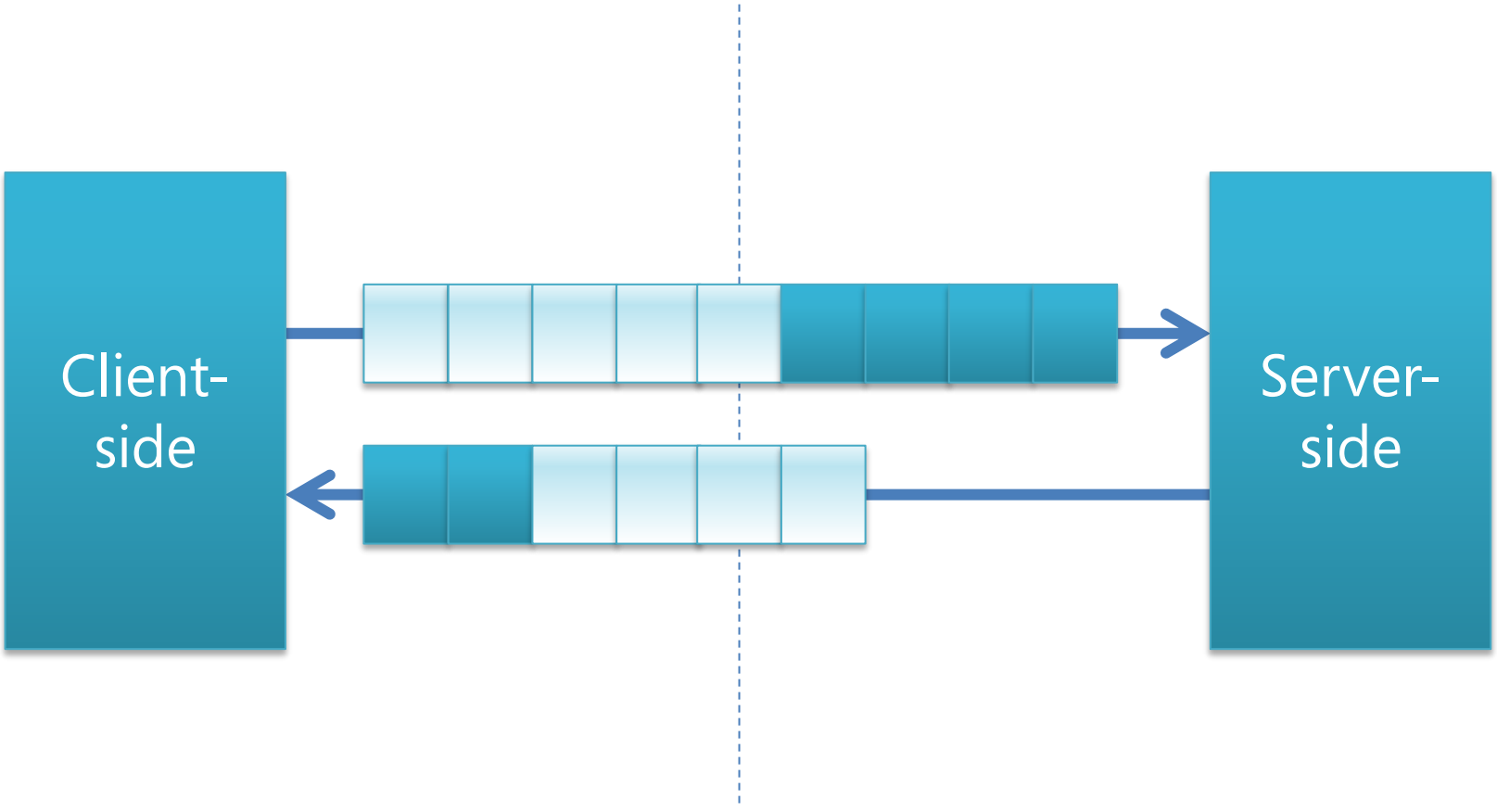
Channel abstraction



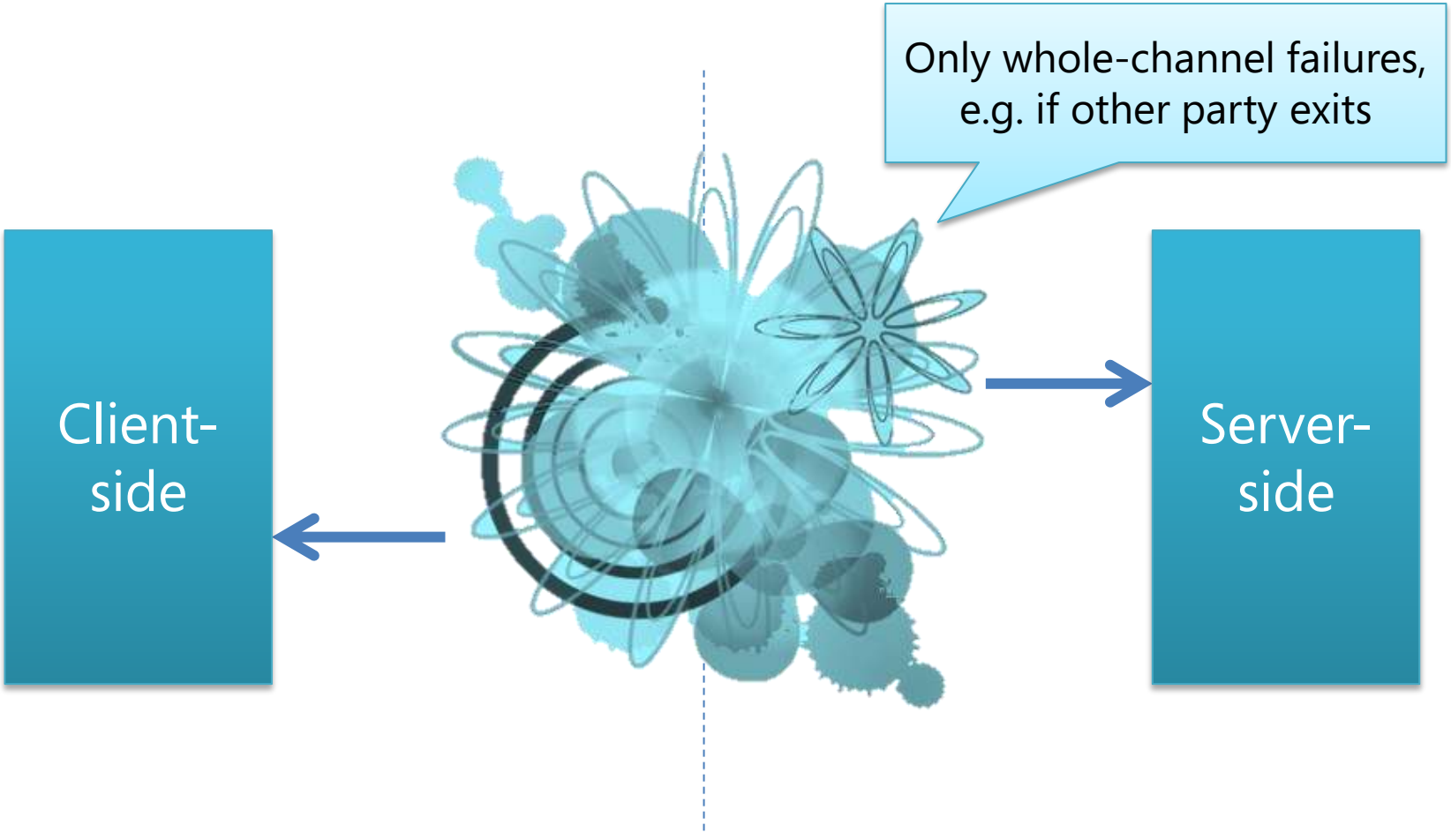
Channel abstraction



Channel abstraction



Channel abstraction



Two back-to-back RPCs

```
static int total_rpc(Echo_binding *b1,
                     Echo_binding *b2,
                     int arg) {
  int result1, result2;
  Echo_ping(b1, arg, &result1);
  Echo_ping(b2, arg, &result2);
  return result1+result2;
}
```

- This looks cleaner but:
  - We've lost the ability to contact multiple servers concurrently
  - We've lost the ability to overlap computation with waiting

Synchronous send/receive

Supporting concurrency

Cancellation

Performance

Adding asynchrony: async, do..finish

```
static int total_rpc(Echo_binding *b1,
                     Echo_binding *b2,
                     int arg) {
  int result1, result2;
  do {
    async Echo_ping(b1, arg, &result1);
    async Echo_ping(b2, arg, &result2);
  } finish;
  return result1+result2;
}
```

Adding asynchrony: async, do..finish

```
static int total_rpc(Echo_binding *b1,
                     Echo_binding *b2,
                     int arg) {
  int result1, result2;
  do {
    async Echo_ping(b1, arg, &result1);
    async Echo_ping(b2, arg, &result2);
  } finish;
  return result1+result2;
}
```

If the async code blocks, then resume after the async

Adding asynchrony: async, do..finish

```
static int total_rpc(Echo_binding *b1,
                     Echo_binding *b2,
                     int arg) {
  int result1, result2;
  do {
    async Echo_ping(b1, arg, &result1);
    async Echo_ping(b2, arg, &result2);
  } finish;
  return result1+result2;
}
```

If the async code blocks, then resume after the async

Wait until all the async work (dynamically) in the do..finish has completed

Example: same-core L4-style RPC

```
static int total_rpc(Echo_binding *b1,
                     Echo_binding *b2,
                     int arg) {
  int result1, result2;
  do {
    async Echo_ping(b1, arg, &result1);



    async Echo_ping(b2, arg, &result2);



  } finish;
  return result1+result2;
}
```

Example: same-core L4-style RPC

```
static int total_rpc(Echo_binding *b1,
                     Echo_binding *b2,
                     int arg) {
  int result1, result2;
  do {
    async Echo_ping(b1, arg, &result1);



    async Echo_ping(b2, arg, &result2);



  } finish;
  return result1+result2;
}
```

Execute to the "ping" call as normal

Example: same-core L4-style RPC

```
static int total_rpc(Echo_binding *b1,
                     Echo_binding *b2,
                     int arg) {
  int result1, result2;
  do {
    async Echo_ping(b1, arg, &result1);



    async Echo_ping(b2, arg, &result2);



  } finish;
  return result1+result2;
}
```

Message send transitions directly to recipient process

Example: same-core L4-style RPC

```
static int total_rpc(Echo_binding *b1,
                     Echo_binding *b2,
                     int arg) {
  int result1, result2;
  do {
    async Echo_ping(b1, arg, &result1);



    async Echo_ping(b2, arg, &result2);



  } finish;
  return result1+result2;
}
```

Response returned to caller

Example: same-core L4-style RPC

```
static int total_rpc(Echo_binding *b1,
                     Echo_binding *b2,
                     int arg) {
  int result1, result2;
  do {
    async Echo_ping(b1, arg, &result1);



    async Echo_ping(b2, arg, &result2);


  } finish;
  return result1+result2;
}
```

Caller proceeds through second call

Example: cross-core RPC

```
static int total_rpc(Echo_binding *b1,
                     Echo_binding *b2,
                     int arg) {
  int result1, result2;
  do {
    async Echo_ping(b1, arg, &result1);



    async Echo_ping(b2, arg, &result2);



  } finish;
  return result1+result2;
}
```

Example: cross-core RPC

```
static int total_rpc(Echo_binding *b1,
                     Echo_binding *b2,
                     int arg) {
  int result1, result2;
  do {
    async Echo_ping(b1, arg, &result1);



    async Echo_ping(b2, arg, &result2);



  } finish;
  return result1+result2;
}
```

First call sends message, this core now idle

Example: cross-core RPC

```
static int total_rpc(Echo_binding *b1,
                     Echo_binding *b2,
                     int arg) {
  int result1, result2;
  do {
    async Echo_ping(b1, arg, &result1);



    async Echo_ping(b2, arg, &result2);



  } finish;
  return result1+result2;
}
```

Resume after async; send second message

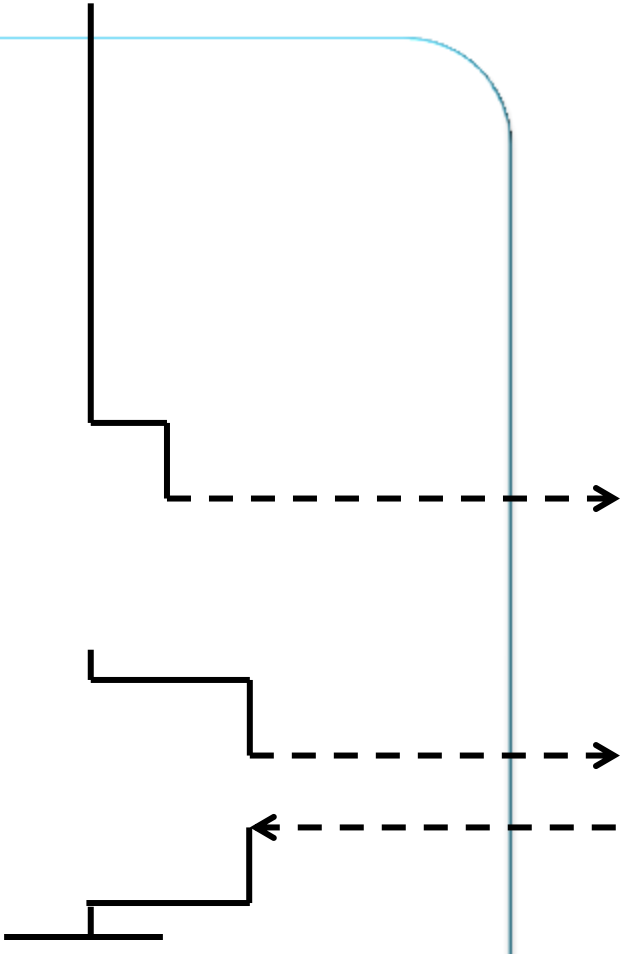
Example: cross-core RPC

```
static int total_rpc(Echo_binding *b1,
                     Echo_binding *b2,
                     int arg) {
  int result1, result2;
  do {
    async Echo_ping(b1, arg, &result1);



    async Echo_ping(b2, arg, &result2);


  } finish;
  return result1+result2;
}
```

Resume after async;
block here until done

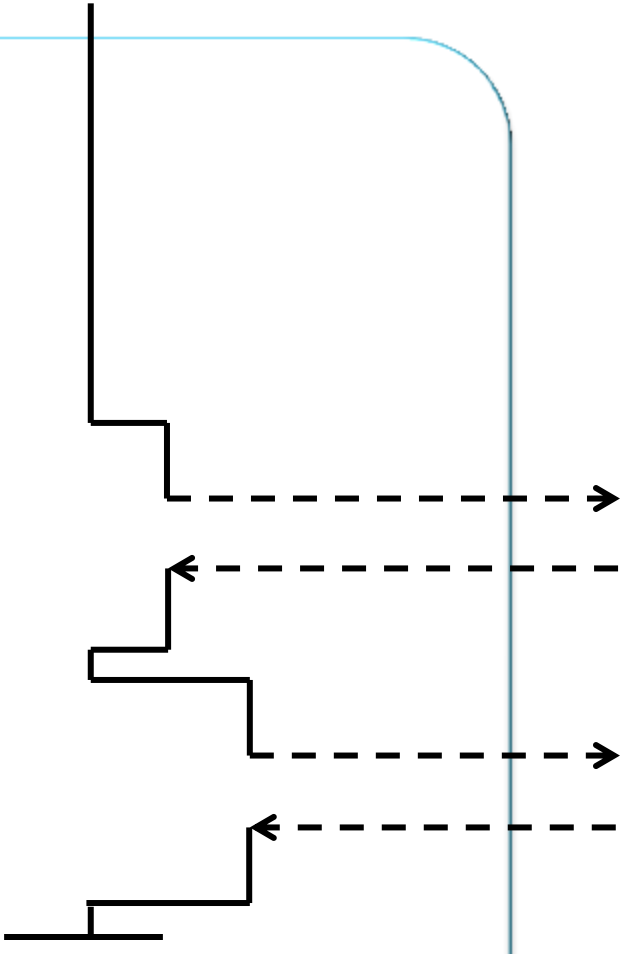
Example: cross-core RPC

```
static int total_rpc(Echo_binding *b1,
                     Echo_binding *b2,
                     int arg) {
  int result1, result2;
  do {
    async Echo_ping(b1, arg, &result1);



    async Echo_ping(b2, arg, &result2);



  } finish;
  return result1+result2;
}
```



Example: cross-core RPC

```
static int total_rpc(Echo_binding *b1,
                     Echo_binding *b2,
                     int arg) {
  int result1, result2;
  do {
    async Echo_ping(b1, arg, &result1);



    async Echo_ping(b2, arg, &result2);



  } finish;
  return result1+result2;
}
```



Example: cross-core RPC

```
static int total_rpc(Echo_binding *b1,
                     Echo_binding *b2,
                     int arg) {
  int result1, result2;
  do {
    async Echo_ping(b1, arg, &result1);


    async Echo_ping(b2, arg, &result2);


  } finish;
  return result1+result2;
}
```

Continue now both responses received

Synchronous send/receive

Supporting concurrency

Cancellation

Performance

Cancellation

- Send a request on n channels
- Receivers vote yes/no
- Wait for $\left\lfloor \frac{n}{2} \right\rfloor + 1$ votes one way or the other

Cancellation

- Se
- Re
- W

```
static bool tally_votes(Vote_binding *b[], int n) {
  int yes=0, no=0;
  int winning = (n/2)+1;
  do {
    for (int i = 0; i < n; i ++) {
      async {
        bool v;
        Vote_get_vote(b[i], &v);
        if (v) { yes++; } else { no++; }
        if (yes>=winning || no >=winning) { ??? }
      }
    }
  } finish;
  return (yes>no);
}
```

Cancellation

- Se
- Re
- W

```
static bool tally_votes(Vote_binding *b[], int n) {
  int yes=0, no=0;
  int winning = (n/2)+1;
  do {
    for (int i = 0; i < n; i ++) {
      async {
        bool v;
        Vote_get_vote(b[i], &v);
        if (v) { yes++; } else { no++; }
        if (yes>=winning || no >=winning) { ??? }
      }
    }
  } finish;
  return (yes>no);
}
```

What should go here?

Cancellation

- Se
- Re
- W

```
static bool tally_votes(Vote_binding *b[], int n) {
  int yes=0, no=0;
  int winning = (n/2)+1;
  do {
    for (int i = 0; i < n; i ++) {
      async {
        bool v;
        Vote_get_vote(b[i], &v);
        if (v) { yes++; } else { no++; }
        if (yes>=winning || no >=winning) { ??? }
      }
    }
  } finish;
  return (yes>no);
}
```

What should go here?

This "finish" must wait for all the enclosed async work…

Cancellation

```
static bool tally_votes(Vote_binding *b[], int n) {
  int yes=0, no=0; int winning = (n/2)+1;
  sync_event s; sync_init(&s);
  do {
    for (int i = 0; i < n; i ++) {
      async {
        bool v;
        if (Vote_get_vote_x(b[i], &v) != CANCELLED) {
          if (v) { yes++; } else { no++; }
          if (yes>=winning || no >=winning) sync_signal(&s);
        }
      }
    }
    sync_wait(&s);
  } cancel;
  return (yes>no);
}
```

Cancellation

Blocking "wait" operation, non-blocking "signal" operation

```
atic bool tally_votes(Vote_binding *b[], int n) {
int yes=0, no=0; int winning = (n/2)+1;
sync_event s; sync_init(&s);
do {
  for (int i = 0; i < n; i ++) {
    async {
      bool v;
      if (Vote_get_vote_x(b[i], &v) != CANCELLED) {
        if (v) { yes++; } else { no++; }
        if (yes>=winning || no >=winning) sync_signal(&s);
      }
    }
  }
  sync_wait(&s);
} cancel;
return (yes>no);
}
```

Cancellation

Blocking "wait" operation, non-blocking "signal" operation

```
atic bool tally_votes(Vote_binding *b[], int n) {
int yes=0, no=0; int winning = (n/2)+1;
sync_event s; sync_init(&s);
do {
  for (int i = 0; i < n; i ++) {
    async {
      bool v;
      if (Vote_get_vote_x(b[i], &v) != CANCELLED) {
        if (v) { yes++; } else { no++; }
        if (yes>=winning || no >=winning) sync_signal(&s);
      }
    }
  }
  sync_wait(&s);
} cancel;
return (yes>no);
}
```

"_x" suffix marks this as a cancelable function

Cancellation

Blocking "wait" operation, non-blocking "signal" operation

```
atic bool tally_votes(Vote_binding *b[], int n) {
int yes=0, no=0; int winning = (n/2)+1;
sync_event s; sync_init(&s);
do {
  for (int i = 0; i < n; i ++) {
    async {
      bool v;
      if (Vote_get_vote_x(b[i], &v) != CANCELLED) {
        if (v) { yes++; } else { no++; }
        if (yes>=winning || no >=winning) sync_signal(&s);
      }
    }
  }
  sync_wait(&s);
} cancel;
return (yes>no);
}
```

"_x" suffix marks this as a cancelable function

We block here waiting for "s" to be signalled

Cancellation

Blocking "wait" operation, non-blocking "signal" operation

```
atic bool tally_votes(Vote_binding *b[], int n) {
int yes=0, no=0; int winning = (n/2)+1;
sync_event s; sync_init(&s);
do {
  for (int i = 0; i < n; i ++) {
    async {
      bool v;
      if (Vote_get_vote_x(b[i], &v) != CANCELLED) {
        if (v) { yes++; } else { no++; }
        if (yes>=winning || no >=winning) sync_signal(&s);
      }
    }
  }
  sync_wait(&s);
} cancel;
return (yes>no);
}
```

"_x" suffix marks this as a cancelable function

We block here waiting for "s" to be signalled

do..cancel requests cancellation of outstanding async work

Cancellation with deferred clean-up

```
...
async {
  bool v;
  if (Echo_tx_get_vote_x(b[i]) == OK) {
    if (Echo_rx_get_vote_x(b[i], &v)) == OK) {
      if (v) { yes++; } else { no++; }
      if (yes>=winning || no >=winning) sync_signal(&s);
    } else {
      push_work(q, [=]{ Echo_tx_apologise(b[i])});
    }
  }
}
...
```

Cancellation with deferred clean-up

Split RPC into explicit send/receive: distinguish two cancellation points

```
...
async {
  bool v;
  if (Echo_tx_get_vote_x(b[i]) == OK) {
    if (Echo_rx_get_vote_x(b[i], &v)) == OK) {
      if (v) { yes++; } else { no++; }
      if (yes>=winning || no >=winning) sync_signal(&s);
    } else {
      push_work(q, [=]{ Echo_tx_apologise(b[i])});
    }
  }
}
...
```

Cancellation with deferred clean-up

Split RPC into explicit send/receive: distinguish two cancellation points

```
...
async {
  bool v;
  if (Echo_tx_get_vote_x(b[i]) == OK) {
    if (Echo_rx_get_vote_x(b[i], &v)) == OK) {
      if (v) { yes++; } else { no++; }
      if (yes>=winning || no >=winning) sync_signal(&s);
    } else {
      push_work(q, [=]{ Echo_tx_apologise(b[i])});
    }
  }
}
...
```

C++ closure to apologies for lost votes

Cancellation with deferred clean-up

```
...
async {
  bool v;
  if (Echo_tx_get_vote_x(b[i]) == OK) {
    if (Echo_rx_get_vote_x(b[i], &v)) == OK) {
      if (v) { yes++; } else { no++; }
      if (yes>=winning || no >=winning) sync_signal(&s);
    } else {
      push_work(q, [=]{ Echo_tx_apologise(b[i])});
    }
  }
}
...
```

Split RPC into explicit send/receive: distinguish two cancellation points

Push clean-up operation to queue if cancelled after tx, before rx

C++ closure to apologies for lost votes

Cancellation with deferred clean-up

Split RPC into explicit send/receive: distinguish two cancellation points

```
...
async {
  bool v;
  if (Echo_tx_get_vote_x(b[i]) == OK) {
    if (Echo_rx_get_vote_x(b[i], &v)) == OK) {
      if (v) { yes++; } else { no++; }
      if (yes>=winning || no >=winning)
    } else {
      push_work(q, [=]{ Echo_tx_apolo
    }
  }
}
...
```

Push clean-up operation to queue if cancelled after tx, before rx

```
// Pseudo-code
do {
  while (true) {
    cleanup_op = pop_work(q);
    async op();
  }
} finish;
```

Cancellation

- Block-structured & co-operative
  - Only triggered from within the same thread
  - Default is cancellation does not occur
  - Exposed via return code
- Cancellation explicit to caller and callee via "_x"
  - Rule: only call a cancellable function within a cancellable function or a do..cancel block
  - Caller needs to know if things they're invoking can be cancelled
  - Callee needs to know that they might be requested to cancel
- Convention:
  - Cancellable function rolls back state
  - RPC stubs discard orphan responses

Synchronous send/receive
Supporting concurrency
Cancellation
Performance

Performance

	Ping-pong latency (cycles)
Using UMP channel directly	931
Using event-based stubs	1134
Synchronous model (client only)	1266
Synchronous model (client and server)	1405
MPI (Visual Studio 2008 + HPC-Pack 2008 SDK)	2780

Ping-pong test

Minimum-sized messages

AMD 4 * 4-core machine

Using cores sharing L3 cache