Libprocess

支撑Mesos的C++并发编程库

MesosCon 2017
Asia

Jay Guo
Benjamin Mahler

Libprocess Overview

- Libprocess is a C++ library for building systems out of *composable* concurrent components
  libprocess是一个C++库，利用“可组合”的并发组件构建系统。

- Mesos is built atop Libprocess, used heavily in production.
  Mesos构建在libprocess之上，在生产环境中大量使用。

- Libprocess has been a great help in making Mesos highly scalable and responsive.
  libprocess使得Mesos具有很好的扩展性和响应度。

Development

- Originally authored by Benjamin Hindman, development now driven by the Mesos project: `3rdparty/libprocess` in github.com/apache/mesos
  原作者是Benjamin Hindman，现在的开发由Mesos社区驱动

- But, treated as a separate project in terms of commits. May be moved out fully from Mesos, but not at the current time
  但是，从代码提交的角度来说，是一个单独的项目。将来也许会从Mesos分离出来作为单独的项目。

Motivation for Libprocess
Libprocess的动机

- Concurrency is <u>hard</u>
  并发是一件很难的事情

  - Not only *correctness*, but also *performance*
    不仅仅出于正确性考量，也因为性能

- We want <u>composable</u> concurrency, in order to **safely** build an **efficient** highly concurrent system
  我们需要<u>可组合</u>的并发，从而**安全地**构建**高效的**高并发的系统

Building Blocks for Concurrent Systems
并发系统的基础成分

- Need to be able to program asynchronously
  需要能够异步编程

Building Blocks for Concurrent Systems
并发系统的基础成分

- Need to be able to program asynchronously
  需要能够异步编程

- Requires a different programming model:
  要求一个不同的编程模型:

```
handle_request(Request r)
{
  doA();
  doB();
  doC();

  send response
}
```

Building Blocks for Concurrent Systems
并发系统的基础成分

- Need to be able to program asynchronously
  需要能够异步编程

- Requires a different programming model:
  要求一个不同的编程模型：

```
handle_request(Request r)
{
    doA();
    doB();
    doC();

    send response
}
```

what if A,B,C take a really long time?
should we tie up the request handling "thread"?
若*A,B,C*运行时间很长，是否需要将请求处理联合起来?

Building Blocks for Concurrent Systems
并发系统的基础成分

- Need to be able to program <u>asynchronously</u>
  需要能够<u>异步</u>编程

- Requires a different programming model:
  要求一个不同的编程模型：

```
handle_request(Request r)
{
    doA();
    doB();
    doC();

    send response
}
```

what if B,C can run in parallel but both depend on A? How do we express that?
如何使得*B,C*可以并行，但都依赖于*A*？如何表示以上逻辑?

Asynchronous Programming
异步编程

- Two schools of thought:
  两种思路：

  1. **Implicit**: Async programming is too hard for programmers. Make it look synchronous, and have it be asynchronous under the covers.
     隐式：从程序员角度来说异步编程过于复杂。让程序看上去是同步的，但内部异步执行。

  2. **Explicit**: Expose asynchronicity directly to programmers.
     显式：把异步特性直接暴露给程序员

Asynchronous Programming
异步编程

1. **Implicit** approach, example from Golang
   隐式的方法，比如Go语言

```
func echo_handler(
    response http.ResponseWriter,
    request *http.Request)
{
    body, error := ioutil.ReadAll(request.Body)
    io.WriteString(w, string(body))
}

func main() {
    http.HandleFunc("/test", test)
    log.Fatal(http.ListenAndServe(":8082", nil))
}
```

Asynchronous Programming
异步编程

1. **Implicit** approach, example from Golang
   隐式的方法，比如Go语言

```
func echo_handler(
    response http.ResponseWriter,
    request *http.Request)
{
    body, error := ioutil.ReadAll(request.Body)
    io.WriteString(w, string(body))
}

func main() {
    http.HandleFunc("/test", test)
    log.Fatal(http.ListenAndServe(":8082", nil))
}
```

} *looks synchronous*

Asynchronous Programming
异步编程

1. **Implicit** approach, example from Golang
   隐式的方法，比如Go语言

```
func echo_handler(
    response http.ResponseWriter,
    request *http.Request)
{
    body, error := ioutil.ReadAll(request.Body)
    io.WriteString(w, string(body))
}

func main() {
    http.HandleFunc("/test", test)
    log.Fatal(http.ListenAndServe(":8082", nil))
}
```

io.ReadCloser

} *looks synchronous*

Asynchronous Programming
异步编程

1. **Implicit** approach, example from Golang
   隐式的方法，比如Go语言

```
func echo_handler(
    response http.ResponseWriter,
    request *http.Request)
{
    body, error := ioutil.ReadAll(request.Body)
    io.WriteString(w, string(body))
}
```

} *looks synchronous*

But, the data is getting asynchronously read from the socket, decoded and placed into the 'Body'. ReadAll reads from the body until it reads EOF.
但是，数据异步地从*socket*中读出，解码并置于*'Body'*。 *ReadAll*从*body*中读取直到*EOF*

Asynchronous Programming
异步编程

1. **Implicit** approach, example from Golang
   隐式的方法，比如Go语言

```
func echo_handler(
    response http.ResponseWriter,
    request *http.Request)
{
    body, error := ioutil.ReadAll(request.Body)
    io.WriteString(w, string(body))
}
```

looks synchronous

This means that the goroutine will “pause” while waiting for data. Like blocking, except that go can run other goroutines in the interim.
这意味着*goroutine*等待数据时需要“暂停”，类似阻塞，只不过等待期间*Go*可以运行其他的*goroutines*

Asynchronous Programming
异步编程

- **Generally**: function calls are a transfer of resources (e.g. execution context, program stack, registers, etc).
  通常意义上：函数调用是资源的转移（比如执行上下文，程序栈，寄存器等等)

Asynchronous Programming
异步编程

- **Generally**: function calls are a transfer of resources (e.g. execution context, program stack, registers, etc).
  通常意义上：函数调用是资源的转移（比如执行上下文，程序栈，寄存器等等）

i.e. how long will I release control of my "thread"?
例如，我将多久释放我对"线程"的控制

Asynchronous Programming
异步编程

- **Generally**: function calls are a transfer of resources (e.g. execution context, program stack, registers, etc).
  通常意义上：函数调用是资源的转移（比如执行上下文，程序栈，寄存器等等）

```
body, error := ioutil.ReadAll(request.Body)
```

↓

*execution context is released for an arbitrary amount of time, **potentially indefinite**!*
执行上下文可能被释放任意时间，**甚至无限**

Asynchronous Programming
异步编程

- **Generally**: function calls are a transfer of resources (e.g. execution context, program stack, registers, etc).
  通常意义上：函数调用是资源的转移（比如执行上下文，程序栈，寄存器等等）

```
body, error := ioutil.ReadAll(request.Body)
```

↓

despite being asynchronous, programming experience is similar to synchronous blocking
虽然是异步执行，编程的方式近似于同步阻塞

Asynchronous Programming
异步编程

- How to cope with the implicit approach?
  如何应对隐式的编程方法?

Asynchronous Programming
异步编程

- How to cope with the implicit approach?
  如何应对隐式的编程方法?

  - For each function you call, understand whether it has implicit asynchronicity and use accordingly.
    对于每一个方法调用，需要理解它是否具备隐式的异步性，并根据特点进行使用

Asynchronous Programming
异步编程

- How to cope with the implicit approach?
  如何应对隐式的编程方法?

  - For each function you call, understand whether it has implicit asynchronicity and use accordingly.
    对于每一个方法调用，需要理解它是否具备隐式的异步性，并根据特点进行使用

  - Or, program defensively! (Run things in a different context to avoid blocking)
    或者，防御性地进行编程！（在不同的上下文中运行以避免阻塞）

Asynchronous Programming
异步编程

- Defensive programming in implicit model is tedious:
  隐式模型中的防御性编程是很繁琐的：

```
func echo_handler(
    response http.ResponseWriter,
    request *http.Request)
{
  channel := make(chan string)

  go func() {
    body, error := ioutil.ReadAll(request.Body)
    channel <- body
  }()

  // Do more work while the body is being read.

  body := <-channel // Now block.
  io.WriteString(w, string(body))
}
```

Asynchronous Programming
异步编程

- Defensive programming in implicit model is tedious:
  隐式模型中的防御性编程是很繁琐的：

```
func echo_handler(
    response http.ResponseWriter,
    request *http.Request)
{
  channel := make(chan string)

  go func() {
    body, error := ioutil.ReadAll(request.Body)
    channel <- body
  }()

  // Do more work while the body is being read.

  body := <-channel // Now block.
  io.WriteString(w, string(body))
}
```

} *avoid blocking*

Asynchronous Programming
异步编程

- Defensive programming in implicit model is tedious:
  隐式模型中的防御性编程是很繁琐的：

```
func echo_handler(
    response http.ResponseWriter,
    request *http.Request)
{
  channel := make(chan string)

  go func() {
    body, error := ioutil.ReadAll(request.Body)
    channel <- body
  }()

  // Do more work while the body is being read.

  body := <-channel // Now block.
  io.WriteString(w, string(body))
}
```

how to handle the error?

how to implement a timeout on the read?

Concurrency Example in Go

```
c1 := make(chan string)
c2 := make(chan string)

go func() { c1 <- doA() }()
go func() { c2 <- doB() }()
```

do A and B in parallel

```
for i := 0; i < 2; i++ {
  select {
    case msg1 := <-c1:
    case msg2 := <-c2:
    case <-time.After(time.Second * 1): // timeout, bail
  }
}

c3 := make(chan int)

go func() { c3 <- doC(msg1, msg2) }()
```

then C

```
select {
  case result := <-c3:
  case <-time.After(time.Second * 1): // timeout, bail
}
```

Concurrency Example in Go

```
c1 := make(chan string)
c2 := make(chan string)

go func() { c1 <- doA() }()
go func() { c2 <- doB() }()

for i := 0; i < 2; i++ {
  select {
    case msg1 := <-c1:
    case msg2 := <-c2:
    case <-time.After(time.Second * 1): // timeout, bail
  }
}

c3 := make(chan int)

go func() { c3 <- doC(msg1, msg2) }()
select {
  case result := <-c3:
  case <-time.After(time.Second * 1): // timeout, bail
}
```

Exercise for the reader: *How can we apply a single timeout rather than two separate timeouts? Difficult!*

Concurrency Example in Go

```
c1 := make(chan string)
c2 := make(chan string)

go func() { c1 <- doA() }()
go func() { c2 <- doB() }()

for i := 0; i < 2; i++ {
  select {
    case msg1 := <-c1:
    case msg2 := <-c2:
    case <-time.After(time.Second * 1): // timeout
  }
}

c3 := make(chan int)

go func() { c3 <- doC(msg1, msg2) }()
select {
  case result := <-c3:
  case <-time.After(time.Second * 1): // timeout
}
```

Exercise for the reader: *How can we apply a single timeout rather than two separate timeouts? Difficult!*

Claim: *Difficult due to lack of composition*

Futures

- Desires:
  需求：
  - explicit asynchronicity
    显式的同步
  - functional composition
    函数组合

Futures

- **Explicit asynchronicity**

Synchronous function:

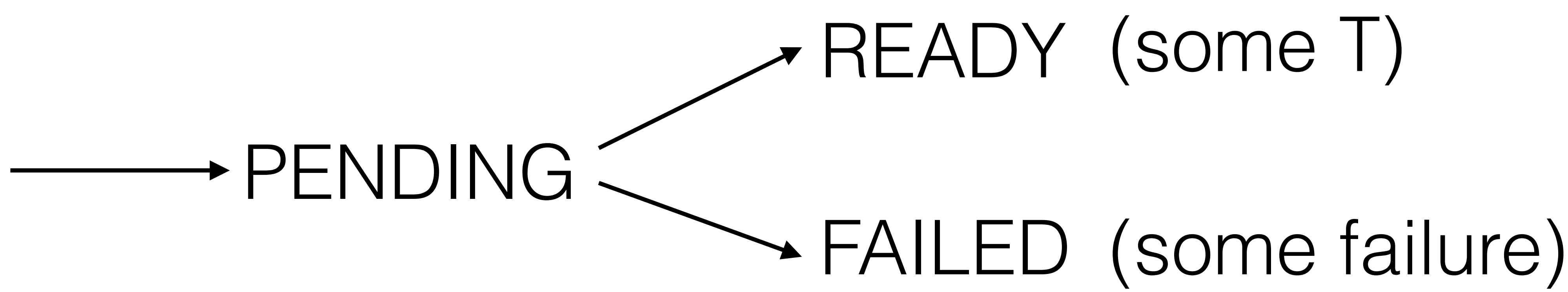
```
T f();
```

Asynchronous function:

```
Future<T> f();
```

Futures

Future<T> state transition



Futures

```
Future<T> future = f();

future.await(); // ANTI-PATTERN in
                // libprocess

if (future.isReady()) {
  T t = future.get();
} else if (future.isFailed()) {
  string failure = future.failure();
}
```

Futures

`Future<T>` is owned by a `Promise<T>`

Client side does not see the Promise

```
Future<T> f = func();
```

Promise performs the transition

```
Future<T> func()
{
  Promise<T> p;
  p.set(T());
  return p.future();
}
```

Futures

- **Futures provide functional composition with .then**
  Futures通过.then提供了函数的组合

```
Future<double> f1 = compute_pi();
Future<double> f2 = f1.then(doubleIt);
Future<string> f3 = f2.then(stringify);

// Or, more simply:

Future<string> f =
  compute_pi()
    .then(doubleIt)
    .then(stringify);
```

Futures

- **Futures provide functional composition with .then**
  Futures通过.then提供了函数的组合

```
Future<string> f =
  compute_pi()
    .then(doubleIt)
    .then(stringify);
```

If any step in the "chain" fails, the failure will propagate into 'f'

Futures

```
Future<string> f =
  compute_pi()
    .then(doubleIt)
    .then(stringify);
```

Which execution context should run the callbacks?

More on this later!

Futures

- Additional features:
  - cancellation via **discard** (client side cancellation request) and **DISCARDED** (terminal state)
  - timeout handling via **after**
  - state specific callbacks via **onReady**, **onFailed**, **onDiscarded**, **onAny**.

Putting it together

```
c1 := make(chan string)
c2 := make(chan string)

go func() { c1 <- doA() }()
go func() { c2 <- doB() }()

for i := 0; i < 2; i++ {
  select {
    case msg1 := <-c1:
    case msg2 := <-c2:
    case <-time.After(time.Second * 1): // timeout
  }
}

c3 := make(chan int)

go func() { c3 <- doC(msg1, msg2) }()
select {
  case result := <-c3:
  case <-time.After(time.Second * 1): // timeout
}
```

Recall golang example from ealier: A and B in parallel, then C. Hard to add a timeout across the two phases.

Putting it together

Future-based approach

```
Future<int> f =
  collect(doA(),doB())
    .then([](tuple<string, string> t) {
      return doC(get<0>(t), get<1>(t));
    }

f = f.after(Seconds(2), [](Future<int> f) {
  f.discard();
  return Failure("timeout");
});

return f;
```

} *A and B, then C*

} *Single timeout for entire computation + **cancellation**!*

Putting it together

Future-based approach

```
Future<int> f =
  collect(doA(),doB())
    .then([](tuple<string, string> t) {
      return doC(get<0>(t), get<1>(t));
    }

f = f.after(Seconds(2), [](Future<int> f) {
  f.discard();
  return Failure("timeout");
});

return f;
```

Assuming that doA, doB, doC are already asynchronous and returning Futures

Putting it together

Future-based approach

```
Future<int> f =
  collect(async(doA), async(doB))
    .then([](tuple<string, string> t) {
      return async([=]() { doC(get<0>(t), get<1>(t)); });
    }

f = f.after(Seconds(2), [](Future<int> f) {
  f.discard();
  return Failure("timeout");
});

return f;
```

If doA, doB, doC are synchronous, can make them asynchronous with 'async'

Putting it together

Future-based approach

```
Future<int> f =
  collect(async(doA), async(doB))
    .then([](tuple<string, string> t) {
      return async([=]() { doC(get<0>(t), get<1>(t)); });
    }

f = f.after(Seconds(2), [](Future<int> f) {
  f.discard();
  return Failure("timeout");
});

return f;
```

How does async work? ***Needs to run it in another execution context.***

Spawn a thread for every async? Too expensive.
async is provided by libprocess, will cover this shortly

Libprocess: Primitives

- actor-like **Process** and **PID** (ala Erlang)
- Local message passing via **dispatch**, **defer** (deferred dispatch), and **delay** (delayed dispatch).
  本地消息传输
- Functional composition via **Futures/Promises**
  函数组合
- Remote message passing via **install**, **send** / monitoring via **link**, **exited** notification.
  远程消息传输

Libprocess: Features

- Asynchronous event loop via libev (or libevent)
  异步事件循环
- Parallel (schedules Processes onto worker threads)
  并行（在worker线程上调度Processes）
- Collection of many asynchronous utilities
  异步的工具集
- Provides a metrics library for monitoring
  提供了一个metrics库进行监控
- Provides testing infrastructure
  提供测试环境
- C++11 (C++14 soon)

Libprocess: Programming Model

- Each Process has a “queue” of incoming “messages”
  每个Process有一个接受的“消息队列”

- Each Process provides an **execution context** (only one thread executing within a Process at a time)
  每一个Process提供一个执行上下文（在一个Process中同一时间只有一个线程在执行）

  - No per Process synchronization!
    没有Process的同步！

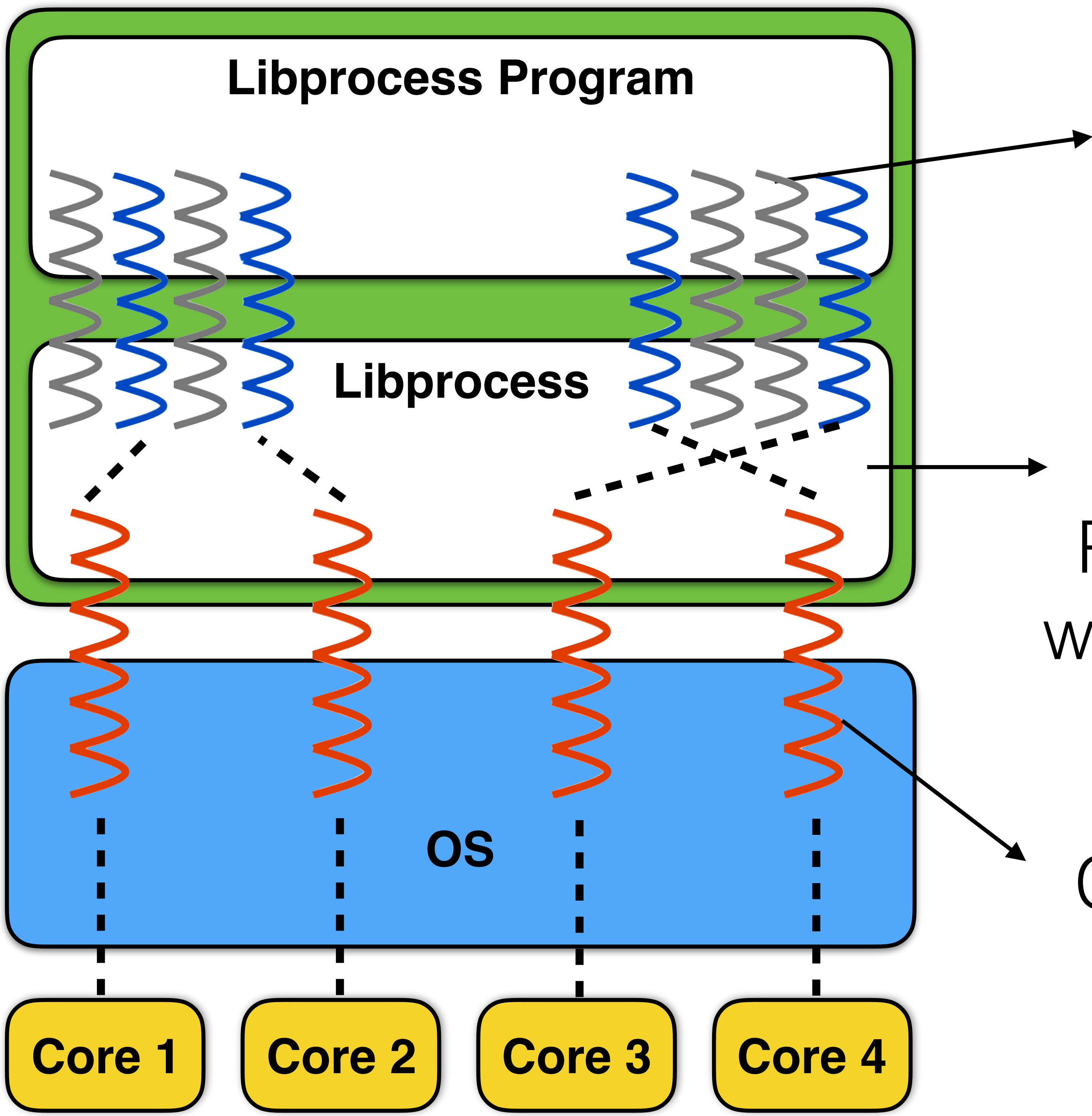
  - Blocking a Process is strictly forbidden!
    禁止阻塞Process！

Libprocess: Programming Model (cont'd)

- No explicit "receive loop" (two cents: "receive loop" is like untyped actor assembly)
  没有显式的"循环接受"

- Processes are more like well typed asynchronous objects (at least locally).
  Processes更像是有类型的异步对象（起码在本地来讲）

Libprocess: Runtime



many Processes:
spawning a process is very cheap (no stack allocation, no thread creation, etc)

libprocess schedules Processes onto threads when Process' queue has messages

Configurable number of worker threads

Process: Lifecycle

```
class MyProcess : public Process<MyProcess> {};

int main()
{
  MyProcess process;

  spawn(process);
  terminate(process);
  wait(process);

  return 0;
}
```

dispatch

```
class QueueProcess : public Process<QueueProcess> {
public:
  void enqueue(int i) { this->i = i; }
  int dequeue() { return this->i; }

private:
  int i;
};

int main() {
  QueueProcess process;
  spawn(process);

  dispatch(process, &QueueProcess::enqueue, 42);

  terminate(process);
  wait(process);
  return 0;
}
```

PID

- Process ID
    - Provides a level of indirection for naming a Process, so that you don't need an actual reference to it (necessary for remote communication!)
      对Process进行了一层封装，这样就不需要实际的引用就可以访问（在远端通讯中是必须的）
    - For local communication, typically a “typed” PID<T>
      本地通讯中，通常是“有类型”的PID<T>
    - For remote communication, typically an “untyped” PID<> (a.k.a. UPID).
      远端通讯中，通常是“无类型”的PID<>（又叫做UPID）

dispatch with PID

```
int main() {
  QueueProcess process;
  PID<QueueProcess> pid = spawn(process);

  dispatch(pid, &QueueProcess::enqueue, 42);

  terminate(pid);
  wait(pid);
  return 0;
}
```

dispatch Future integration

```
class QueueProcess : public Process<QueueProcess> {
public:
  void enqueue(int i) { this->i = i; }
  int dequeue() { return this->i; }

private:
  int i;
};

int main() {
  QueueProcess process;
  PID<QueueProcess> pid = spawn(process);

  dispatch(pid, &QueueProcess::enqueue, 42);

  Future<int> i = dispatch(pid, &QueueProcess::dequeue);

  terminate(pid);
  wait(pid);
}
```

syntax sugar: Process “Wrapper”

```
template <typename T>
class Queue {
public:
  Queue() { spawn(q); }
  ~Queue() { terminate(q); wait(q); }

  void enqueue(T t) { dispatch(q, &QueueProcess::enqueue, t); }
  Future<T> dequeue() { return dispatch(q, &QueueProcess::dequeue);

private:
  QueueProcess<T> q;
};

int main() {
  Queue<int> queue;
  queue.enqueue(42);
  queue.dequeue()
    .then([](int i) {
      // use it
    });
}
```

callback invocation

```
template <typename T>
class Queue {
public:
  Queue() { spawn(q); }
  ~Queue() { terminate(q); wait(q); }

  void enqueue(T t) { dispatch(q, &QueueProcess::enqueue, t); }
  Future<T> dequeue() { return dispatch(q, &QueueProcess::dequeue);

private:
  QueueProcess<T> q;
};

int main() {
  Queue<int> queue;
  queue.enqueue(42);
  queue.dequeue()
    .then([](int i) {
      // use it
    });
}
```

When should this callback get invoked?

Using which execution context?

callback invocation

- Either invoke the callback…
  - synchronously using the current thread
  - asynchronously using a different thread, but which thread?

synchronous callback invocation
同步回调函数

+ can be more efficient when callback is trivial
回调函数轻量的时候，更加高效

— can't access state of the "callback owner" without synchronization (hard to compose)
没有同步的话，无法访问"回调函数所有者"的状态（导致难以组合）

— hard to reason about performance since the current thread may be delayed for an indefinite amount of time! (not to mention loss of registers, cache misses, etc?)
性能很难评估，因为当前线程可能被延迟很长时间执行！（还要考虑寄存器损失，缓存未命中等）

asynchronous callback invocation
异步回调函数

- semantics of “charging” the “callback owner”

defer

- Provides a **deferred dispatch** on a Process

```
class SomeProcess : public Process<SomeProcess> {
public:
  void merge() {
    queue.dequeue()
      .then(defer(self(), [this](int i) {
        // use it within context of SomeProcess
      }));
  }
};
```

async

- Turns a synchronous function into an asynchronous one

```
T func();

Future<T> f = async(func);
```

- Works by spawning a one-off Process, runs ‘func’ in this Process. (Could also use a dedicated async Process, or a pool of async Processes, etc).

一些构建工具

Owned<T> & Shared<T>

- Encapsulate smart pointers for Memory Management
- unique_ptr vs shared_ptr
- Shared<T> enforces `const` access
- share Owned<T> via `share()`
- own Shared<T> via `own()`, which returns a Future. *One of them succeeds and others fail

Owned<T> & Shared<T>

```
Try<Owned<Provisioner>> _provisioner =
  Provisioner::create(flags_, secretResolver);

if (_provisioner.isError()) {
  return Error("Failed to create provisioner: " + _provisioner.error());
}

Shared<Provisioner> provisioner = _provisioner.get().share();
```

Abstraction

- Async Queue
- Async Mutex
- Async Pipe
- Subprocess

Async Queue

- Concurrent Queue implementation with `std::queue`
- serialized using `std::atomic_flag`
- Empty? `get` a Future!
- Next `put` fulfills that Future without enqueue

Async Queue

```
Queue<string> q;
Future<string> get1 = q.get(); // get1 would be PENDING
q.put("Hello");                // get1 is READY

q.put("MesosCon");
Future<string> get2 = q.get(); // get2 is READY immediately
```

Async Mutex

- Asynchronously `lock()` so it's not blocked
- queued Futures for `lock()` attempts

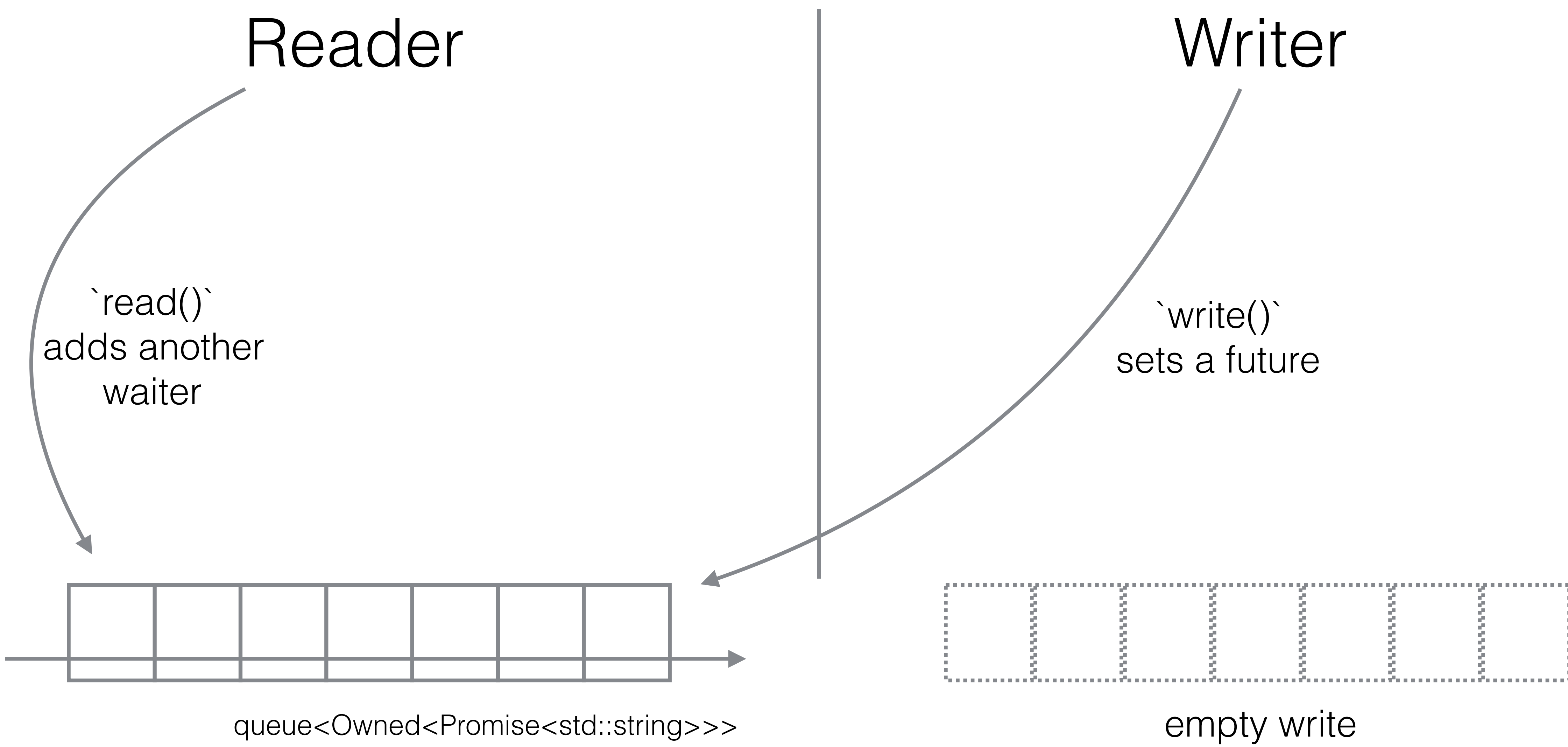
Async Mutex

```
mutex.lock()
  .then(defer(self(), [this]() {
    // critical section here
  }))
  .onAny(lambda::bind(&Mutex::unlock, mutex));
```

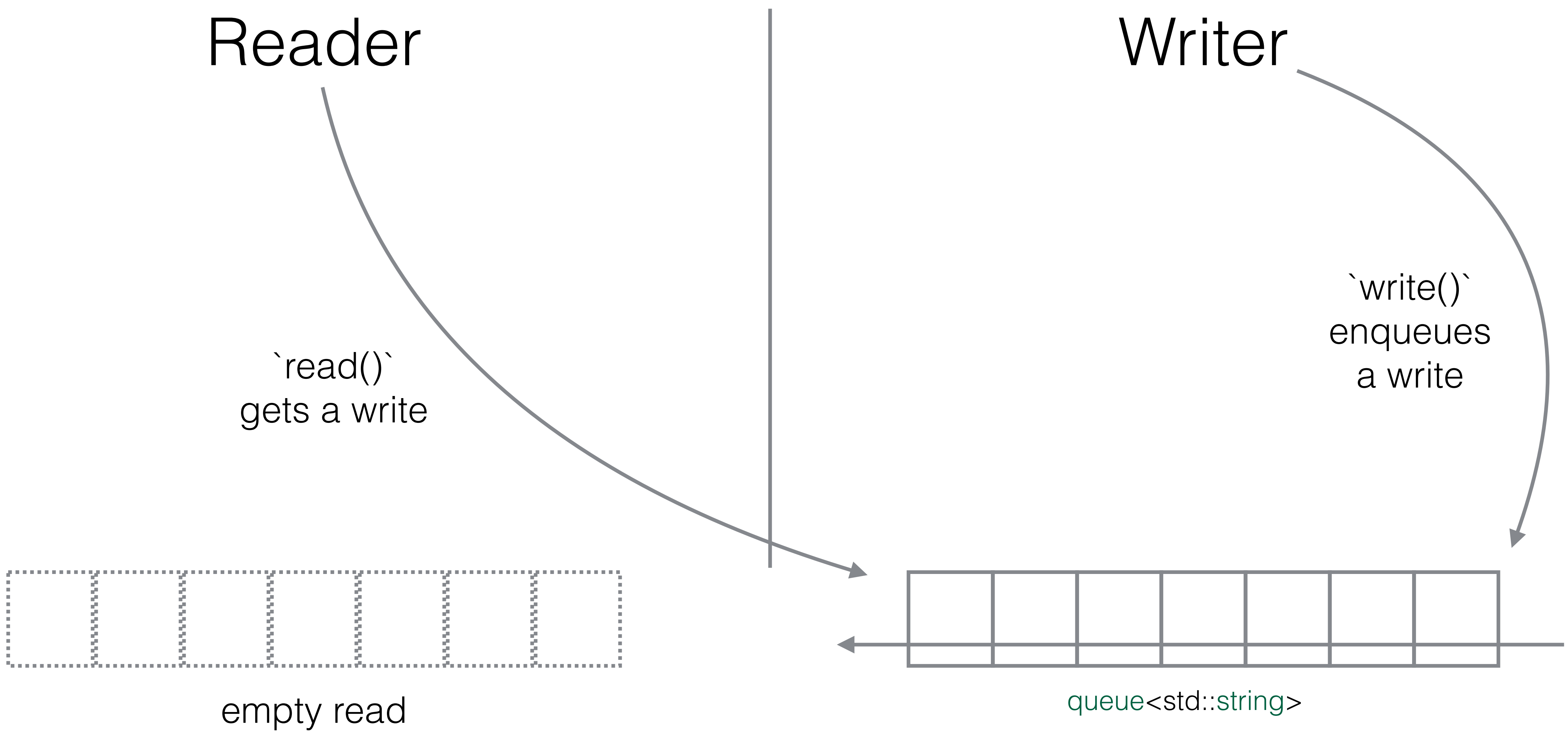
Async Pipe

- In-memory
- data is read until EOF
- Used for streaming client/server request/response

Async Pipe


Reader
Writer
`read()` adds another waiter
`write()` sets a future
queue<Owned<Promise<std::string>>>
empty write

Async Pipe



Reader
Writer
`read()` gets a write
`write()` enqueues a write
empty read
queue<std::string>

Subprocess

- Represents a `fork()` exec()ed subprocess
- Often used to execute a command, e.g. docker pull, launch a process in containerized context

Subprocess

```
Try<Subprocess> s = subprocess(
    "echo 'hello' && sleep 10",
    Subprocess::FD(STDIN_FILENO),
    Subprocess::FD(outFd.get()),
    Subprocess::FD(STDERR_FILENO));

s.get().status()
  .then(…)
  .after(
      Seconds(5),
      [](…) {
        // Kill the process
});
```

Subprocess

```
Try<Subprocess> s = subprocess(
    "echo 'hello' && sleep 10",
    Subprocess::FD(STDIN_FILENO),
    Subprocess::FD(outFd.get()),
    Subprocess::FD(STDERR_FILENO));

s.get().status()
  .then(…)
  .after(
      Seconds(5),
      [](…) {
        // Kill the process
});
```

Redirect input/output/err

Subprocess

```
Try<Subprocess> s = subprocess(
    "echo 'hello' && sleep 10",
    Subprocess::FD(STDIN_FILENO),
    Subprocess::FD(outFd.get()),
    Subprocess::FD(STDERR_FILENO));

s.get().status()
  .then(…)
  .after(
      Seconds(5),
      [](…) {
        // Kill the process
});
```

Redirect input/output/err

Chain in Futures

Subprocess

```
Try<Subprocess> s = subprocess(
    "echo 'hello' && sleep 10",
    Subprocess::FD(STDIN_FILENO),
    Subprocess::FD(outFd.get()),
    Subprocess::FD(STDERR_FILENO));

s.get().status()
  .then(…)
  .after(
      Seconds(5),
      [](…) {
        // Kill the process
});
```

Redirect input/output/err

Chain in Futures

Set timeout on the process

Test infrastructure

- Clock
- Message filtering & intercepting
- Await

Clock

- timeouts get exercised without actually waiting that long
- time based events get triggered reliably
- pause, advance, settle, resume

Clock

```
Clock::pause();

// Register agents, subscribe frameworks, etc

// Trigger a batch allocation to make sure all resources are
// offered out again.
Clock::advance(masterFlags.allocation_interval);

// Settle to make sure all offers are received.
Clock::settle();

// Some other stuff

Clock::resume();
```

AWAIT_*

- Block waiting till future is fulfilled, for 15s

AWAIT_*

```
Clock::pause();

// Start master

Future<Nothing> addSlave;

// Start agent

Clock::advance();

AWAIT_READY(addSlave);
```

Block waiting & assert

Message filtering and intercepting

- Expecting certain types of message?
- Need to spoof a message to simulate certain scenario?

Message filtering and intercepting

```
Future<ReregisterSlaveMessage> reregisterSlaveMessage =
  DROP_PROTOBUF(
      ReregisterSlaveMessage(),
      slave.get()->pid,
      master.get()->pid);

AWAIT_READY(reregisterSlaveMessage);

// Spoof the message here

process::post(
    slave.get()->pid,
    master.get()->pid,
    spoofedReregisterSlaveMessage);
```

Message filtering and intercepting

```
Future<ReregisterSlaveMessage> reregisterSlaveMessage =
  DROP_PROTOBUF(
      ReregisterSlaveMessage(),
      slave.get()->pid,
      master.get()->pid);

AWAIT_READY(reregisterSlaveMessage);

// Spoof the message here

process::post(
    slave.get()->pid,
    master.get()->pid,
    spoofedReregisterSlaveMessage);
```

hijack the message delivered to master

Message filtering and intercepting

```
Future<ReregisterSlaveMessage> reregisterSlaveMessage =
  DROP_PROTOBUF(
      ReregisterSlaveMessage(),
      slave.get()->pid,
      master.get()->pid);

AWAIT_READY(reregisterSlaveMessage);

// Spoof the message here

process::post(
    slave.get()->pid,
    master.get()->pid,
    spoofedReregisterSlaveMessage);
```

hijack the message delivered to master

deliver spoofed message

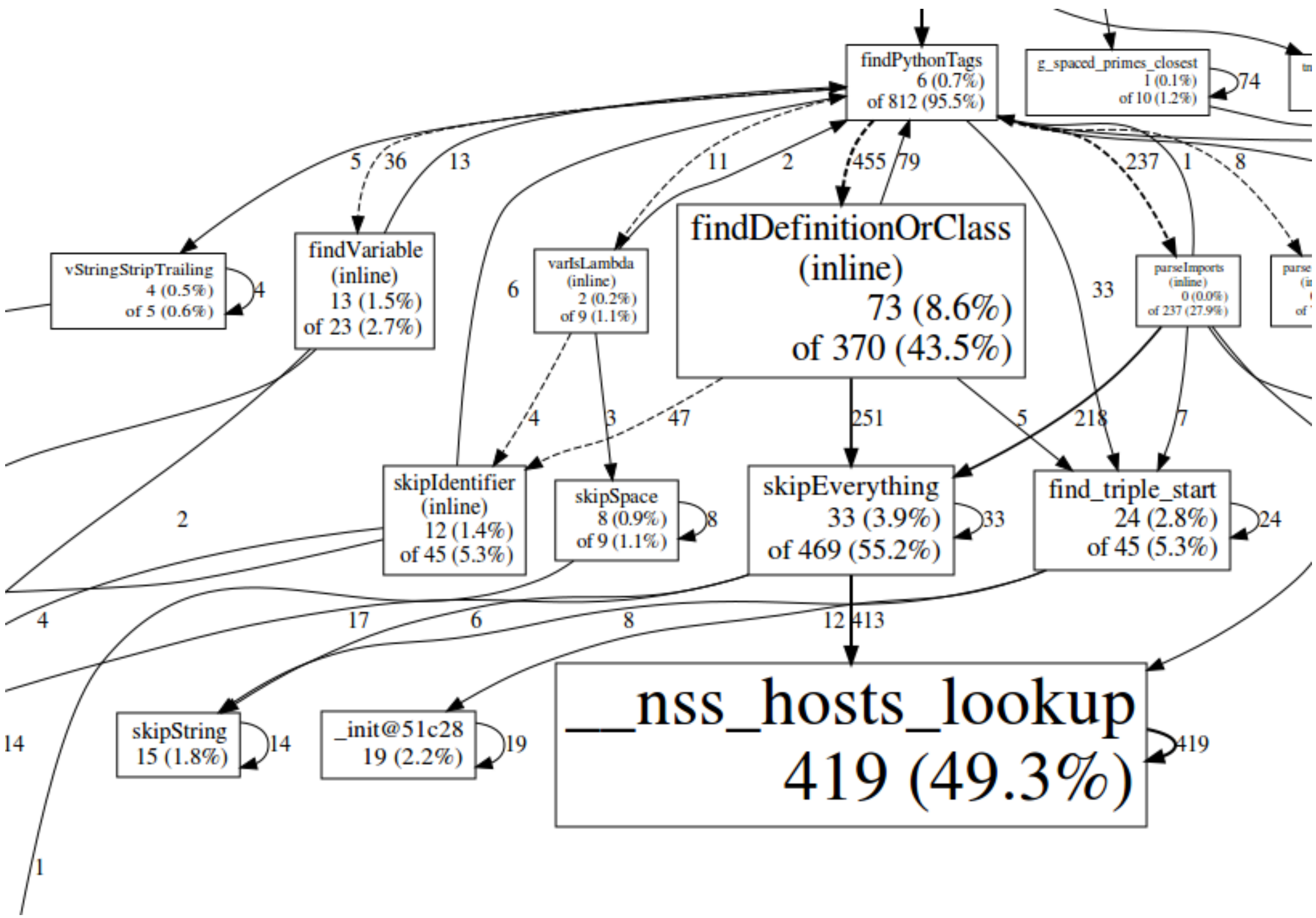
Configurations

- LIBPROCESS_IP:PORT
  - useful on a multi-homed box
- LIBPROCESS_ADVERTISE_IP:PORT
  - useful if IP:PORT is not directly reachable from other nodes, e.g. NAT
- LIBPROCESS_NUM_WORKER_THREADS
  - prevent overwhelming # of threads on a powerful machine, e.g. ppc64le
- LIBPROCESS_ENABLE_PROFILER
  - used when profiling libprocess

Profiling & Metrics

- Built-in metrics library
- Endpoint exposing metrics snapshot
- Built-in cpu profiler using gperftools

Profiling & Metrics



Future Work

- **Lots** of optimization work!
- HTTP 2 / gRPC support
- More asynchronous abstractions (e.g. Stream<T>)
- C++14 / C++17
- Better documentation / examples