LIBPROCESS

A C++ Library for Concurrency

ANKUR VERMA

Definition

Actor style
message-passing
programming model

Primitives

- Process and PID
- Functional composition via promises/futures
- Local messaging via *dispatch, delay, and defer*
- Remote messaging via *send, route, and install*

Features

- Asynchronous event-loop (via libev)
- Parallel
- Integrated HTTP support (transport protocol)
- Testing infrastructure

Table of Concepts

- Async defines a function template for asynchronously executing function closures. It provides their results as Futures.
- dispatch schedules a method for asynchronous execution
- defer is a way to *defer* a dispatch.
- delay instead of dispatching for execution right away, it allows it to be scheduled after a certain time duration.

```
1
2
3   using namespace process;
4   void foo()
5   {
6     ProcessBase process;
7     spawn(process);
8     Deferred<void(int)> deferred = defer(
9         process,
10        [](int i) {
11          // Invoked _asynchronously_ using `process` as the
12          // execution context.
13        });
14    Promise<int> promise;
15    promise.future().then(deferred);
16    promise.future().then([](int i) {
17      // Invoked synchronously from the execution context of
18      // the thread that completes the future!
19    });
20    // Executes both callbacks synchronously, which _dispatches_
21    // the deferred lambda to run asynchronously in the execution
22    // context of `process` but invokes the other lambda immediately.
23    promise.set(42);
24    terminate(process);
25  }
26
27
28  As another example, consider this excerpt from the Mesos project's src/master/master.cpp:
29
30  // Start contending to be a leading master and detecting the current leader.
31  // NOTE: `.onAny` passes the relevant future to its callback as a parameter, and
32  // `lambda::_1` facilitates this when using `defer`.
33
34  contender->contend()
35    .onAny(defer(self(), &Master::contended, lambda::_1));
```

- A Future acts as the read-side of a result which might be computed asynchronously.

  Futures can be created in numerous ways: awaiting the result of a method call with defer, dispatch, and delay or as the read-end of a promise.

- A promise is an object that can fulfill a futures, i.e. assign a result value to it.

- A PID provides a level of indirection for naming a process without having an actual reference (pointer) to it (necessary for remote processes).

- A process is an actor, effectively a cross between a thread and an object.

- A route installs an http endpoint onto a process.

Examples: Without Libprocess

```
1  #include<future>
2
3  using namespace std;
4
5  void factorial(int N)
6  {
7     int res=1;
8
9     for(int i=1; i<=N; i++)
10       res*=i;
11
12    cout<<"Result is:" << res <<endl;
13
14 }
15
16 int main()
17 {
18    std::thread t1(factorial , 4);
19
20    t1.join();
21
22    return 0;
23
24 }
25
26
```

Now , I want to return the result from child thread to parent process

```
1   #include<future>
2
3   using namespace std;
4
5   void factorial(int N , int& x)
6   {
7     int res=1;
8
9     for(int i=1; i<=N; i++)
10      res*=i;
11
12    cout<<"Result is:" << res <<endl;
13    x = res;
14
15  }
16
17  int main()
18  {
19    int x;
20    std::thread t1(factorial , 4, std::ref(x));
21
22    t1.join();
23
24    return 0;
25
26  }
```

Now since x is a shared variable between child and parent process (add mutex) and variable x should be set in child first then accessible by parent (add cond for wait and notify)

```cpp
#include<future>

using namespace std;
std::mutex mu;
std::condition_variable cond;

void factorial(int N , int& x)
{
  int res=1;

  for(int i=1; i<=N; i++)
    res*=i;

  cout<<"Result is:" << res <<endl;
  x = res;

}

int main()
{
  int x;
  std::thread t1(factorial , 4, std::ref(x));  // thread class

  t1.join();

  return 0;

}
```

Avoid threads

```cpp
#include<future>

using namespace std;

void factorial(int N )
{
  int res=1;

  for(int i=1; i<=N; i++)
    res*=i;

  cout<<"Result is:" << res <<endl;
  return res;

}

int main()
{
   int x;
  // std::thread t1(factorial , 4, std::ref(x); //child thread
  std::future<int> fu = std::async(factorial , 4); // async, a function may or mayn't create a new thread
  x = fu.get()

  // t1.join();

  return 0;

}
```

Async vs deffered

```cpp
#include<future>

using namespace std;

void factorial(int N )
{
  int res=1;

  for(int i=1; i<=N; i++)
    res*=i;

  cout<<"Result is:" << res <<endl;
  return res;

}

int main()
{
   int x;
 // std::thread t1(factorial , 4, std::ref(x));
  std::future<int> fu = std::async(std::launch::deffered, factorial , 4); //async will not create a new thread rather it will defer and wait till fu.get is called which will call factorial func in same thread
  std::future<int> fu = std::async(std::launch::async, factorial , 4);//creates a new thread]
  std::future<int> fu = std::async(std::launch::deffered | std::launch::async, factorial , 4); //anything can happen similiar to std::future<int> fu = std::async(factorial , 4);
  x = fu.get()

 // t1.join();

  return 0;

}
```

Now we can also write a code in which we define that value will be passed from parent to child, promising that value will sent for sure

```
#include<future>

using namespace std;

void factorial(std::future<int>& f )
{
  int res=1;
  N = f.get();
  for(int i=1; i<=N; i++)
    res*=i;

  cout<<"Result is:" << res <<endl;
  return res;

}

int main()
{
  int x;
  std::promise<int> p;
  std::future<int> f = p.get_future();

  std::future<int> fu = std::async(std::launch::async, factorial , std::ref(f));


  //do smoething
  //sleep for sometime
  std::this_thread::sleep_for(chrono::milliseconds(20));

  p.set_value(4);
   x = fu.get()
  return 0;

}
```

Now , somehow if promise is not kept and forgot to set value

```cpp
#include<future>

using namespace std;

void factorial(std::future<int>& f )
{
  int res=1;
  N = f.get();    //std::future_errc::broken_promise
  for(int i=1; i<=N; i++)
    res*=i;

  cout<<"Result is:" << res <<endl;
  return res;

}

int main()
{
  int x;
  std::promise<int> p;
  std::future<int> f = p.get_future();

  std::future<int> fu = std::async(std::launch::async, factorial , std::ref(f));


  //do smoething
  //sleep for sometime
  std::this_thread::sleep_for(chrono::milliseconds(20));

  //p.set_value(4);
  //x = fu.get()

  return 0;

}
```

- Now to avoid exception , set exception on promise
- p.set_exception(std::make_exception_ptr(std::runtime_error("to err is human")));

Now suppose you have 10 threads to call the same factorial(), so you cant share the same future but f.get() can't be possible (Broadcasting : shared_future)

```cpp
#include<future>

using namespace std;

void factorial(std::shared_future<int> f )
{
  int res=1;
  N = f.get();    //std::future_errc::broken_promise
  for(int i=1; i<=N; i++)
    res*=i;

  cout<<"Result is:" << res <<endl;
  return res;

}

int main()
{
  int x;
  std::promise<int> p;
  std::future<int> f = p.get_future();
  std::shared_future<int> sf = f.share();
  std::future<int> fu = std::async(std::launch::async, factorial , sf);
  std::future<int> fu2 = std::async(std::launch::async, factorial , sf);
  std::future<int> fu3 = std::async(std::launch::async, factorial , sf);


  //do smoething
  //sleep for sometime
  std::this_thread::sleep_for(chrono::milliseconds(20));

  //p.set_value(4);
  //x = fu.get()

  return 0;

}
```

Using libprocess library

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

  MyProcess() {}

  virtual ~MyProcess() {}

  void set_Promise(int val)
  {
    p.set(val);
    cout<< "Value set" << endl;
  }

  Future<int> factorial( )
  {

      int res=1;
      int N =  print_promise().get();
      for(int i=1; i<=N; i++)
      res*=i;
      cout<<"Result is:" << res <<endl;

   return Promise<int>(res).future();
  }

  Future<int> print_promise()
  {
      return p.future();
  }

private:

  Promise<int> p;

};

int main()
{
  MyProcess myprocess;

  PID<MyProcess> myp_pid = spawn(&myprocess);
  cout <<"myprocess is created..... " << myp_pid << endl;

   process::dispatch(myp_pid, &MyProcess::set_Promise, 4);

   Future<int> f = process::dispatch(myp_pid, &MyProcess::print_promise);
   cout <<"Promise set : = " << f.get() << endl;

  Future<int> res =  process::dispatch(myp_pid, &MyProcess::factorial);
  // process::dispatch(myp_pid, &MyProcess::set_Promise, 4);
  cout <<" res is = " << res.get() << endl;

  return 0;

}
```

Server_client1.cpp

```cpp
#include <chrono>
#include <iostream>
#include <thread>

#include <process/delay.hpp>
#include <process/dispatch.hpp>
#include <process/future.hpp>
#include <process/process.hpp>


#include <stout/json.hpp>
#include <stout/numify.hpp>


using std::cerr;
using std::cout;
using std::endl;

using std::chrono::seconds;

using process::Future;
using process::Promise;

using process::http::Request;
using process::http::OK;
using process::http::InternalServerError;

class ServerProcess : public process::Process<ServerProcess> {

public:

  ServerProcess() : ProcessBase("server") {}

 void initialize() {
  route(
      "/logout",
      "logout from server",
      [this] (Request request) {
        this->logout_server();
        return OK("Logged out, can't access the endpoint       ");
  });


 }

```

Contd.

```
  Future<bool> disconnected() {
    cout << "Shall we disconnect? " << endl;
    return shouldQuit.future();
  }

  void logout_server() {
    cout << "Logging out from server..." << endl;
    this->shouldQuit.set(true);
  }
private:
  Promise<bool> shouldQuit;
};

int main() {
  int retCode;
  ServerProcess simpleProcess;
  process::PID<ServerProcess> pid = process::spawn(simpleProcess);

  cout << "Running Server on http://" << process::address().ip << ":"
       << process::address().port << "/server" << endl
       << "Use /logout to terminate server..." << endl;
  cout << "Waiting for it to terminate..." << endl;

  Future<bool> disconnect  = simpleProcess.disconnected();
  disconnect.await();

  // wait for the server to complete the request
  std::this_thread::sleep_for(seconds(2));

  if (!disconnect.get()) {
    cerr << "The server encountered an error and is exiting now" << endl;
    retCode = EXIT_FAILURE;
  } else {
    cout << "Disconnected" << endl;
    retCode = EXIT_SUCCESS;
  }
  cout << "retCode = " << retCode <<endl;

  process::terminate(simpleProcess);
  process::wait(simpleProcess);

  return 0;
}
```

Output using
curl http://172.31.43.242:33385/server/logout

```
ubuntu@ip-172-31-43-242:~/ankur/libs$ ./serv
Running Server on http://172.31.43.242:33385/server
Use /logout to terminate server...
Waiting for it to terminate...
Shall we disconnect?
Logging out from server...
Disconnected
retCode = 0
```

Server_client2.cpp

```cpp
#include <chrono>
#include <iostream>
#include <thread>

#include <process/delay.hpp>
#include <process/dispatch.hpp>
#include <process/future.hpp>
#include <process/process.hpp>
#include <process/http.hpp>

#include <stout/json.hpp>
#include <stout/numify.hpp>


using std::cerr;
using std::cout;
using std::endl;
using std::string;
using std::chrono::seconds;

using process::Future;
using process::Promise;
using namespace process::http;
using process::http::Request;
using process::http::OK;
using process::http::InternalServerError;

class ServerProcess : public process::Process<ServerProcess> {

public:

  ServerProcess() : ProcessBase("server") {}

 void initialize() {

  route(
        "/api/v1/scheduler",
        "hit",
        [=] (Request request) {
         return this->res();
   });
 }
```

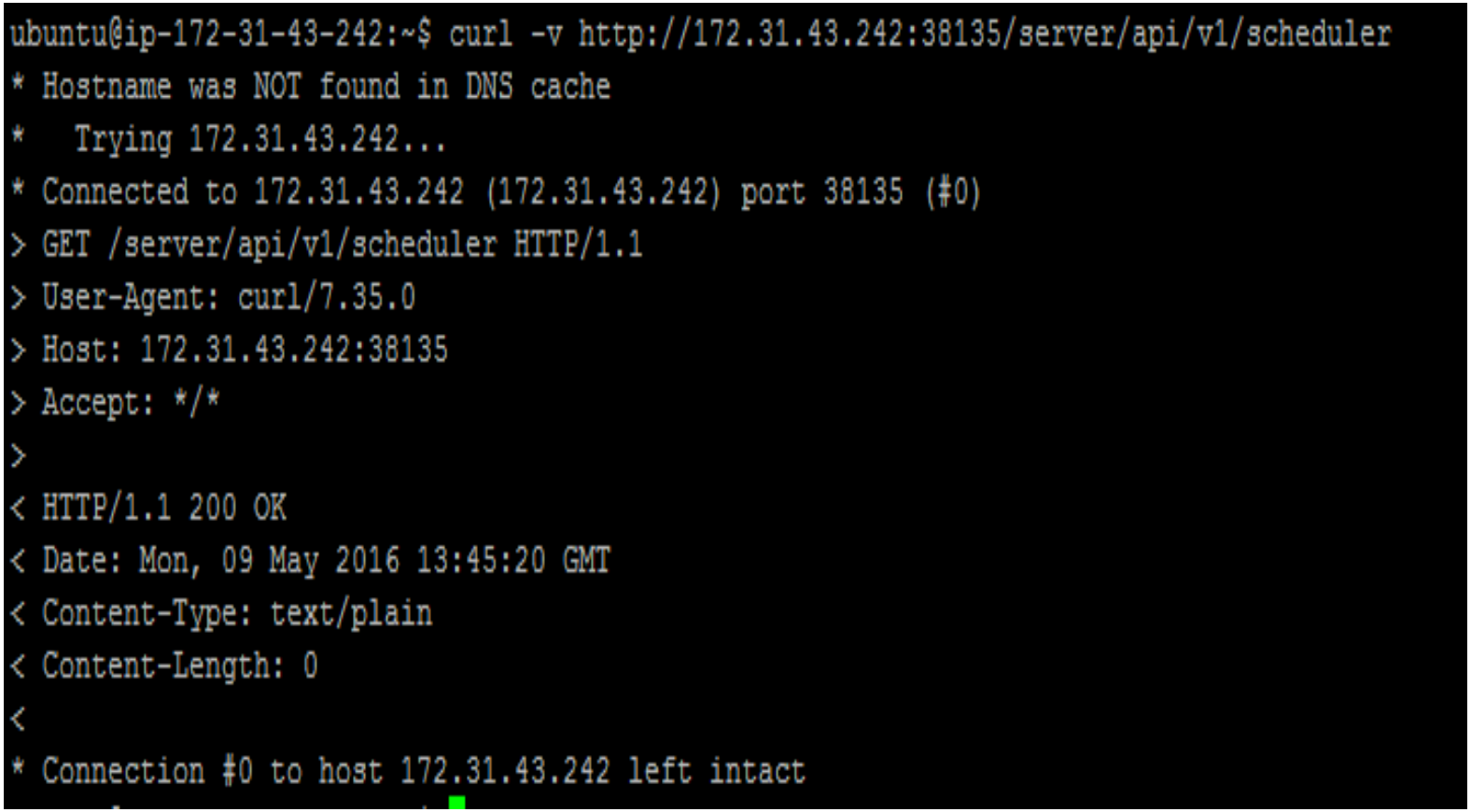
Contd.

```
  Future<bool> getResponse() {
    cout << "Shall we get response? " << endl;
    return shouldGet.future();
  }
   Future<Response> res() {
        string body = "... vars here ...";
        OK response;
        response.headers["Content-Type"] = "text/plain";
        std::ostringstream out;
        out << body.size();
        response.headers["Content-Length"] = out.str();
        response.body = body;

    this->shouldGet.set(true);
  }
private:
  Promise<bool> shouldGet;
};
int main() {
  int retCode;
  ServerProcess simpleProcess;
  process::PID<ServerProcess> pid = process::spawn(simpleProcess);

  cout << "Running Server on http://" << process::address().ip << ":"
       << process::address().port << "/server" << endl
       << "Use /api/v1/scheduler to get response from server..." << endl;
  cout << "Waiting to get..." << endl;

  Future<bool> getR  = simpleProcess.getResponse();
  getR.await();
  // wait for the server to complete the request
  std::this_thread::sleep_for(seconds(2));

  if (!getR.get()) {
    cerr << "The server encountered an error and is exiting now" << endl;
    retCode = EXIT_FAILURE;
  } else {
    cout << "Response in progress" << endl;
    retCode = EXIT_SUCCESS;
  }
  cout << "retCode = " << retCode <<endl;

  process::terminate(simpleProcess);
  process::wait(simpleProcess);
```

Output using
curl –v http://172.31.43.242:33385/server/api/v1/scheduler

```
ubuntu@ip-172-31-43-242:~/ankur/libs$ ./serv
Running Server on http://172.31.43.242:44079/server
Use /api/v1/scheduler to get response from server...
Waiting to get...
Shall we get response?
Response in progress
retCode = 0
```

```
ubuntu@ip-172-31-43-242:~$ curl -v http://172.31.43.242:44079/server/api/v1/scheduler
* Hostname was NOT found in DNS cache
*   Trying 172.31.43.242...
* Connected to 172.31.43.242 (172.31.43.242) port 44079 (#0)
> GET /server/api/v1/scheduler HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 172.31.43.242:44079
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Mon, 09 May 2016 13:23:44 GMT
< Content-Length: 17
< Content-Type: text/plain
<
* transfer closed with 17 bytes remaining to read
* Closing connection 0
curl: (18) transfer closed with 17 bytes remaining to read
```


ubuntu@ip-172-31-43-242:~$ curl -v http://172.31.43.242:38135/server/api/v1/scheduler
* Hostname was NOT found in DNS cache
*   Trying 172.31.43.242...
* Connected to 172.31.43.242 (172.31.43.242) port 38135 (#0)
> GET /server/api/v1/scheduler HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 172.31.43.242:38135
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Mon, 09 May 2016 13:45:20 GMT
< Content-Type: text/plain
< Content-Length: 0
<
* Connection #0 to host 172.31.43.242 left intact

THANK YOU

Q & A