UNIVERSITÄT
BERN

7. Liveness and Asynchrony

Oscar Nierstrasz

Roadmap

- Asynchronous invocations
- Simple Relays
  - —Direct invocations
  - —Thread-based messages
  - —Command-based messages
- Tail calls
- Early replies
- Futures
- JUC (java.util.concurrent)

Roadmap

- **Asynchronous invocations**
- Simple Relays
  - —Direct invocations
  - —Thread-based messages
  - —Command-based messages
- Tail calls
- Early replies
- Futures
- JUC (java.util.concurrent)

Pattern: Asynchronous Invocations

Intent: Avoid waiting for a request to be serviced by *decoupling sending from receiving*.

Applicability

- When a host object can distribute services amongst multiple helper objects.
- When an object does not immediately need the result of an invocation to continue doing useful work.
- When invocations that are logically asynchronous, regardless of whether they are coded using threads.
- During refactoring, when classes and methods are split in order to increase concurrency and reduce liveness problems.

Asynchronous Invocations — template

Asynchronous invocation typically looks like this:

```
abstract class AbstractHost implements Host {
   public void service() {
      pre();                  // code to run before invocation
      invokeHelper();         // the invocation
      during();               // code to run in parallel
      post();                 // code to run after completion
   }
...
}
```

```
// A host provides a
service
public interface Host {
   public void service();
}
```

We will see many variants of this basic pattern. The helper may run in its own thread. There may or may not be a need to obtain a result from the helper.

Asynchronous Invocations — design steps

Consider the following issues:

Does the Host need results back from the Helper?	Not if, e.g., the Helper returns results directly to the Host's caller!
Can the Host process new requests while the Helper is running?	Might depend on the kind of request ...
Can the Host do something while the Helper is running?	i.e., in the `during()` code
Does the Host need to synchronize pre-invocation processing?	i.e., if `service()` is guarded or if `pre()` updates the Host's state
Does the Host need to synchronize post-invocation processing?	i.e., if `post()` updates the Host's state
Does post-invocation processing only depend on the Helper's result?	... or does the host have to wait for other conditions?
Is the same Helper always used?	Is a new one generated to help with each new service request?

Roadmap

- Asynchronous invocations
- **Simple Relays**
  - —Direct invocations
  - —Thread-based messages
  - —Command-based messages
- Tail calls
- Early replies
- Futures
- JUC (java.util.concurrent)

Simple Relays — three variants

A relay method obtains all its functionality by delegating to the helper, without any `pre()`, `during()`, or `post()` actions.

- ***Direct invocations:***
  - —Invoke the Helper directly, but without synchronization
- ***Thread-based messages:***
  - —Create a new thread to invoke the Helper
- ***Command-based messages:***
  - —Pass the request to another object that will run it

Relays are commonly seen in Adaptors.

Roadmap

- Asynchronous invocations
- Simple Relays
  - **Direct invocations**
  - Thread-based messages
  - Command-based messages
- Tail calls
- Early replies
- Futures
- JUC (java.util.concurrent)

Variant: Direct invocations

```
public class HostDirectRelay implements Host {
  // NB: helper is also immutable, so unsynchronized
  protected final Helper helper = new CountingHelper();

  public void service() { // unsynchronized!
    invokeHelper();          // stateless method
  }

  protected void invokeHelper() {
    helper.help();           // unsynchronized!
  }
}
```

Asynchrony is achieved by avoiding synchronization.

The Host is free to accept other requests, while the Host's caller must wait for the reply.

In this pattern there is no `pre`, `post` or `during` code, so it may seem as though there is no asynchrony. Indeed, there is none *within* the current service, but the point is that *other clients* may call the same service asynchronously (since there is no need for synchronization).

In all the other examples we will see asynchrony within the running service.

Direct invocations ...

If `helper` is mutable, it can be protected with an accessor:

```
public class HostDirectRelaySyncHelper implements Host {
   protected Helper helper;
   public void service() { invokeHelper(); }
   protected void invokeHelper() {
      helper().help();     // partially unsynchronized!
   }
   protected synchronized Helper helper() {
      return helper;
   }
   public synchronized void setHelper(String name) {
      helper = new NamedHelper(name);
   }
}
```

Roadmap

- Asynchronous invocations
- Simple Relays
  - Direct invocations
  - **Thread-based messages**
  - Command-based messages
- Tail calls
- Early replies
- Futures
- JUC (java.util.concurrent)

Variant: Thread-based messages

The invocation can be performed within a new thread:

```
public class HostWithHelperThread implements Host {
   ...
   protected void invokeHelper() {
      new Thread() {
         public void run() {
            helper.help();
         }
      }.start();
   }
   ...
}
```

Thread-based messages ...

The cost of evaluating Helper.help() should outweigh the overhead of creating a thread!

- If the Helper is a *daemon* (loops endlessly)
- If the Helper does *I/O*
- Possibly, if *multiple helper methods* are invoked

Typical application: web servers

Here there is clearly asynchrony *within* the running service as a new thread is spawned to run the helper. Creating a new thread imposes a certain overhead, however, so the task of the helper must be important enough to compensate for the overhead.

A *daemon* is an kind of server thread that runs continuously, for example, performing a service, responding to requests, or monitoring events.

Input/output is almost always much *slower* than simple computation, so it can make sense to *wrap any i/o activity into a helper* (e.g., writing to a file, waiting for user input, requesting input from a network device).

A web server is a good example. For each HHTP request, a server thread may be spawned to handle the request, thus freeing up the server to handle another request.

Thread-per-message Gateways

The Host may construct a new Helper to service each request.

```
public class FileIO {
   public void writeBytes(String file, byte[] data) {
      new Thread (new FileWriter(file, data)).start();
   }
   public void readBytes(...) { ... }
}
class FileWriter implements Runnable {
   private String nm_;              // hold arguments
   private byte[] d_;
   public FileWriter(String name, byte[] data) { ... }
   public void run() { ... }    // write to file ...
}
```

In this example, writing output to a file is performed by a dedicated "`FileWriter`" object within a new thread.

NB: This is skeleton code only, not a running example. Obviously there is an assumption here that *no synchronization is needed* to access the file (unique file names, no file readers). An example could be that of writing log files named after unique timestamps.

Roadmap

- Asynchronous invocations
- Simple Relays
  - —Direct invocations
  - —Thread-based messages
  - —**Command-based messages**
- Tail calls
- Early replies
- Futures
- JUC (java.util.concurrent)

Variant: Command-based messages

The Host can also put a *Command object* in a queue for another object that will invoke the Helper:

```
public class HostEventQueue implements Host {
… protected void invokeHelper() {
      EventQueue.invokeLater(new Runnable() {
        public void run() { helper.help(); }
      }); }
}
```

Command-based forms are especially useful for:

> *scheduling* of helpers (i.e., by *pool* of threads)

> *undo and replay* capabilities

> transporting messages over *networks*

In our web server example, it may be a bad idea to create an unbounded number of helper threads for each incoming request. A way to address this is to use a *fixed pool of threads*. Requests are turned into `Command` objects that are queued and served by the first available helper thread.

Command objects are also used in scenarios where commands may be explicitly undone or even replayed, for example, editing commands in a textual or graphical editor.

Roadmap

- Asynchronous invocations
- Simple Relays
  - —Direct invocations
  - —Thread-based messages
  - —Command-based messages
- **Tail calls**
- Early replies
- Futures
- JUC (java.util.concurrent)

Tail calls

Applies when the helper method is the *last* statement of a service. Only `pre()` code is synchronized.

```
public class TailCallSubject extends Observable {
  protected Observer observer = new Observer() { ... };
  protected double state;
  public void updateState(double d) {   // unsynchronized
    doUpdate(d);                        // partially synchronized
    sendNotification();                 // unsynchronized
  }
  protected synchronized void doUpdate(double d) {
    state = d;
  }
  protected void sendNotification() {
    observer.update(this, state);
  }
}
```

NB: The host is immediately available to accept new requests

Here the helper task is to notify an observer. The actual update (i.e., the pre action) is synchronized, but the notification is not.

There is no `during` action and no `post` action.

Here, as in the case of the simple relay, asynchrony is with respect to new incoming requests that can be served even as the observer is being notified.

Tail calls with new threads

Alternatively, the tail call may be made in a separate thread:

```
public synchronized void updateState(double d) {
 state = d;
 new Thread() {
  public void run() {
   observer.update(TailCallSubject.this, state);
  }
 }.start();
}
```

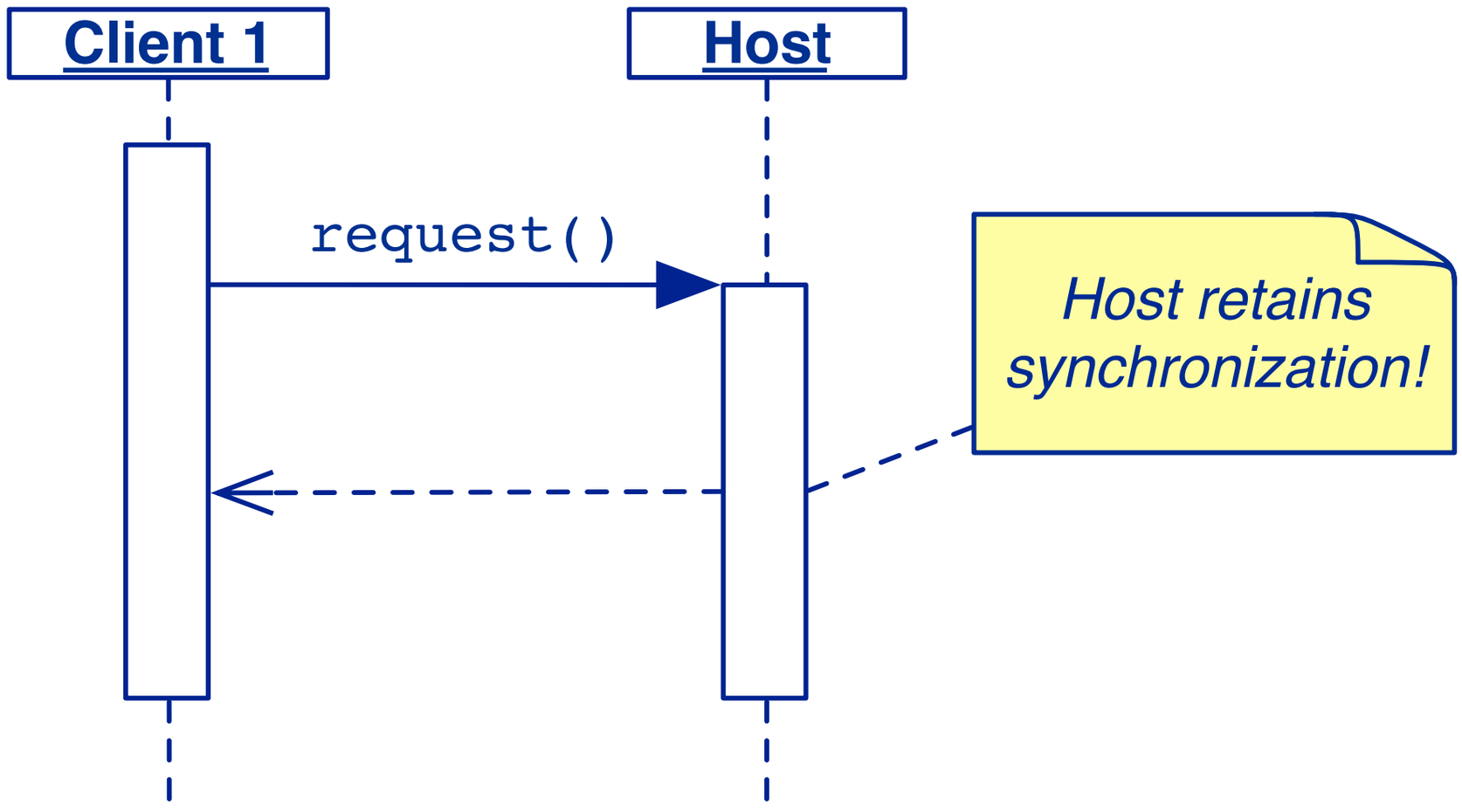
This solution would allow the original caller to proceed immediately, without waiting for the helper to complete. In the case of a single notification, this is unlikely to bring much gain, but if there is a long list of hundreds of observers, it might make a small difference.

Roadmap

- Asynchronous invocations
- Simple Relays
  - —Direct invocations
  - —Thread-based messages
  - —Command-based messages
- Tail calls
- **Early replies**
- Futures
- JUC (java.util.concurrent)

Early Reply

Early reply allows a host to perform useful activities *after returning a result* to the client:

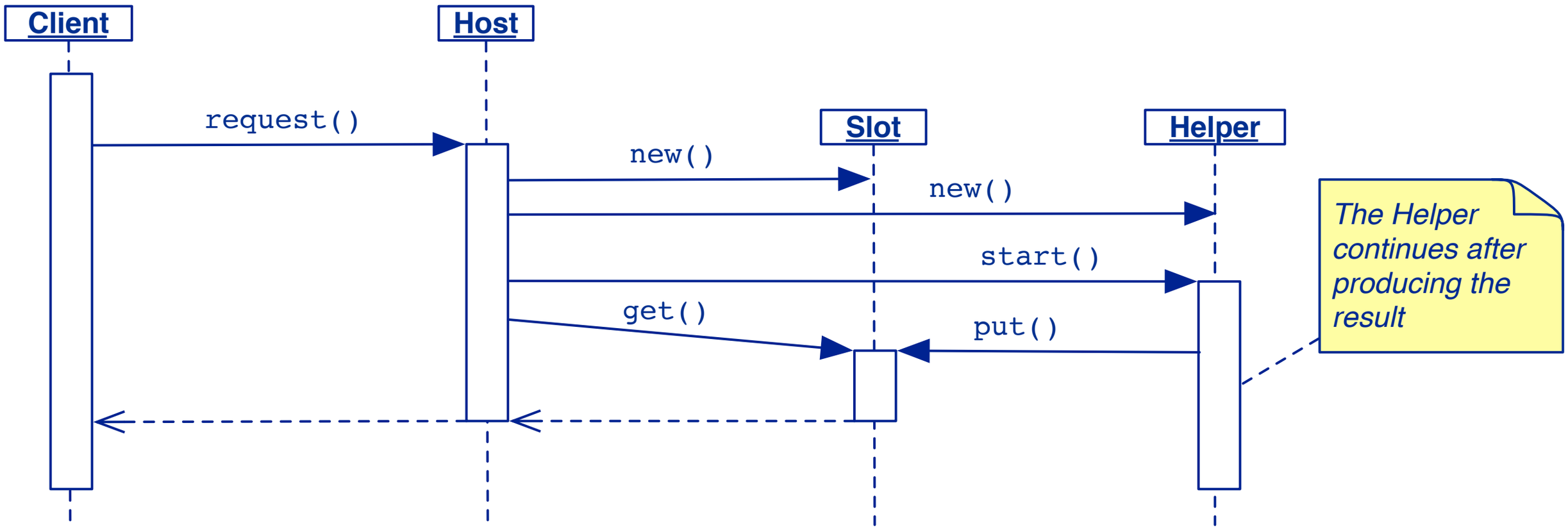


Early reply is a built-in feature in some programming languages.
It can be easily simulated when it is not a built-in feature.

In a typical scenario, the host must perform some cleanup activity after computing and returning the result to the client. The client and the cleanup thread may then proceed concurrently.

Simulating Early Reply

A one-slot buffer can be used to pick up the reply from a helper thread:



A one-slot buffer is a simple abstraction that can be used to implement many higher-level concurrency abstractions ...

The difficulty here is that the host must somehow wait for the helper to compute the return result, but without blocking on a call to the helper (which would be purely synchronous). Instead the host waits on the one-slot buffer to be filled with the reply result from the helper. Once the helper deposits the result, it is free to continue with its cleanup action.

The advantage of this solution is that the client is entirely unaware of the interaction protocol between the host and the helper.

Early Reply in Java

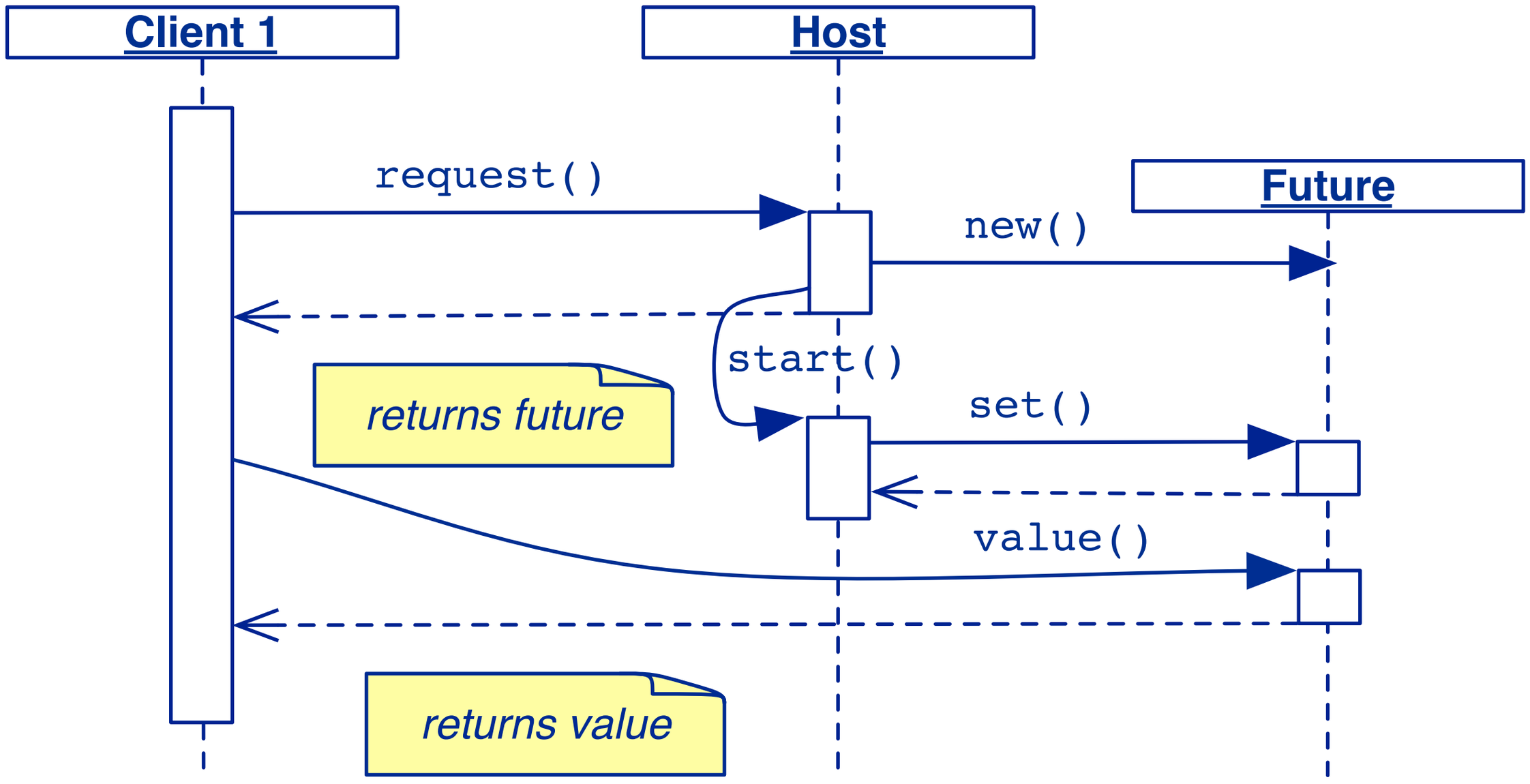
```
public class EarlyReplyDemo { ...
   public Object service() {          // unsynchronized
      final Slot reply = new Slot();
      final EarlyReplyDemo host = this;
      new Thread() {                  // Helper
         public void run() {
            synchronized (host) {
               reply.put(host.compute());
               host.cleanup();        // retain lock
         } }
      }.start();
      return reply.get();             // early reply
   } ...
}
```


Roadmap

- Asynchronous invocations
- Simple Relays
  - —Direct invocations
  - —Thread-based messages
  - —Command-based messages
- Tail calls
- Early replies
- **Futures**
- JUC (java.util.concurrent)

Futures

Futures allow a client to continue in parallel with a host until the future value is needed:



Futures differ from early replies in that they allow the *client* to carry out other work while the host computes the requested "future value".

A future is like a "ticket" for work that is being carried out. When you need the result, you "cash in the ticket". If the future has already been computed, then you immediately obtain its value, otherwise you then wait until it is ready.

In the scenario diagram, a `Future` is similar to a one-slot buffer, except that its value is only set once, and getting the value does not consume it.

A Future Class

```
abstract class Future<Result,Argument> {
   private Result result;                          // initially null
   public Future(final Argument arg) {
      new Thread() {
         public void run() { setResult(computeResult(arg)); }
      }.start();
   }
   abstract protected Result computeResult(Argument arg);
   public synchronized void setResult(Result val) {
      result = val;
      notifyAll();
      return;
   }
   public synchronized Result result() {
      while (result == null) {
         try { wait(); }
         catch (InterruptedException e) { }
      }
      return result;
   }
}
```


There are many possible ways to implement futures. In this version, `Future` is an abstract class with an abstract `computeResult` method that must be defined in a subclass. When the future is instantiated, a new thread is created to compute the result. The result method waits to return the requested result until is has been successfully deposited by the `computeResult` method.

Using Futures in Java

Without special language support, the client must explicitly request a `result()` from the future object.

```
Future<Integer,Integer> f = new Future<Integer,Integer>(n) {
   protected synchronized Integer computeResult(Integer n) {
      return fibonacci(n);
   }
   // slow, naive algorithm to force long compute times ;-)
   public int fibonacci(int n) {
      if (n<2) { return 1; }
      else { return fibonacci(n-1) + fibonacci(n-2); }
   }
};
int val = f.result();
```

Here a concrete subclass of `Future` is defined as an *anonymous inner class*.

Here we see the key drawback of futures when they are not defined as a language feature: the client of the future needs to be aware of the future and *must explicitly request the result* when it is needed. With early replies, on the other hand, clients need take no special action, and may be completely unaware of the use of early replies by the host.

Roadmap

- Asynchronous invocations
- Simple Relays
  - —Direct invocations
  - —Thread-based messages
  - —Command-based messages
- Tail calls
- Early replies
- Futures
- **JUC (java.util.concurrent)**

java.util.concurrent

Executors

- Executor
- ExecutorService
- ScheduledExecutorService
- Callable
- Future
- ScheduledFuture
- Delayed
- CompletionService
- ThreadPoolExecutor
- ScheduledThreadPoolExecutor
- AbstractExecutorService
- Executors
- FutureTask
- ExecutorCompletionService

Queues

- BlockingQueue
- ConcurrentLinkedQueue
- LinkedBlockingQueue
- ArrayBlockingQueue
- SynchronousQueue
- PriorityBlockingQueue
- DelayQueue

Concurrent Collections

- ConcurrentMap
- ConcurrentHashMap
- CopyOnWriteArray{List,Set}

Synchronizers

- CountDownLatch
- Semaphore
- Exchanger
- CyclicBarrier

Locks: java.util.concurrent.locks

- Lock
- Condition
- ReadWriteLock
- AbstractQueuedSynchronizer
- LockSupport
- ReentrantLock
- ReentrantReadWriteLock

Atomics: java.util.concurrent.atomic

- Atomic[Type]
- Atomic[Type]Array
- Atomic[Type]FieldUpdater
- Atomic{Markable,Stampable}Reference

`java.util.concurrent` is a library of concurrency abstractions that can significantly reduce the complexity of programming concurrent applications. It includes many abstractions that we have already seen.

Here we will only provide a very general overview.

Key Functional Groups

- Executors, Thread pools and Futures
  - —Execution frameworks for asynchronous tasking
- Concurrent Collections:
  - —Queues, blocking queues, concurrent hash map, …
  - —Data structures designed for concurrent environments
- Locks and Conditions
  - —More flexible synchronization control
  - —Read/write locks
- Synchronizers: Semaphore, Latch, Barrier
  - —Ready made tools for thread coordination
- Atomic variables
  - —The key to writing lock-free algorithms

The Executor Framework

- Framework for asynchronous task execution
- Standardize asynchronous invocation
  - —Framework to execute `Runnable` and `Callable` tasks
    - *`Runnable: void run()`*
    - *`Callable<V>: V call() throws Exception`*
- Separate submission from execution policy
  - —Use `anExecutor.execute(aRunnable)`
  - —Not `new Thread(aRunnable).start()`
- Cancellation and shutdown support
- Usually created via Executors factory class
  - —Configures flexible `ThreadPoolExecutor`
  - —Customize shutdown methods, before/after hooks, saturation policies, queuing

The Executor framework provides a higher-level way to manage concurrent tasks than directly using threads. In addition to the `Runnable` interface, it also introduces the `Callable` interface, which can return a result.

Instead of directly creating threads, a programmer will ask an "executor" to run tasks.

Executor

- Decouple submission policy from task execution

```
public interface Executor {
   void execute(Runnable command);
}
```

- Code which submits a task doesn't have to know in what thread the task will run
  - —Could run in the calling thread, in a thread pool, in a single background thread (or even in another JVM!)
  - —Executor implementation determines execution policy
    - Execution policy controls resource utilization, overload behavior, thread usage, logging, security, etc
    - Calling code need not know the execution policy

ExecutorService

- Adds lifecycle management
- ExecutorService supports both graceful and immediate shutdown

```
public interface ExecutorService extends Executor {
   void shutdown();
   List<Runnable> shutdownNow();
   boolean isShutdown();
   boolean isTerminated();
   boolean awaitTermination(long timeout, TimeUnit unit);
   // ...
}
```

- Useful utility methods too
  - `<T> T invokeAny(Collection<Callable<T>> tasks)`
    - Executes the given tasks returning the result of one that completed successfully (if any)
  - Others involving Future objects

FutureTask

```
public FutureTask<Integer> service (final int n) {
   FutureTask<Integer> future =
      new FutureTask<Integer> (
         new Callable<Integer>() {
            public Integer call() {
               return new Integer(fibonacci(n));
            }
      });
   new Thread(future).start(); // or use an Executor
   return future;
}
```


JUC provides a generic implementation of Futures, parameterized by Callable or Runnable services.

Whereas the `Future` class shown earlier required a subclass to define the abstract `computeResult` method, a `FutureTask` object is parameterized by a `Callable` object. The basic idea, however, is the same. A client still needs to explicitly request the future result when it is needed.

Creating Executors

- Sample Executor implementations from Executors
- newSingleThreadExecutor
  - —A pool of one, working from an unbounded queue
- newFixedThreadPool(int N)
  - —A fixed pool of N, working from an unbounded queue
- newCachedThreadPool
  - —A variable size pool that grows as needed and shrinks when idle
- newScheduledThreadPool
  - —Pool for executing tasks after a given delay, or periodically

Locks and Synchronizers

- `java.util.concurrent` provides generally useful implementations
  - —`ReentrantLock`, `ReentrantReadWriteLock`
  - —`Semaphore`, `CountDownLatch`, `Barrier`, `Exchanger`
  - —Should meet the needs of most users in most situations
    - Some customization possible in some cases by subclassing
- Otherwise `AbstractQueuedSynchronizer` can be used to build custom locks and synchronizers
  - —Within limitations: `int` state and FIFO queuing
- Otherwise build from scratch
  - —Atomics
  - —Queues
  - —LockSupport for thread parking/unparking

What you should know!

- *What general form does an asynchronous invocation take?*
- *When should you consider using asynchronous invocations?*
- *In what sense can a direct invocation be “asynchronous”?*
- *Why (and how) would you use inner classes to implement asynchrony?*
- *What is “early reply”, and when would you use it?*
- *What are “futures”, and when would you use them?*
- *How can you implement futures and early replies in Java?*

Can you answer these questions?

- *Why might you want to increase concurrency on a single-processor machine?*
- *Why are servers commonly structured as thread-per-message gateways?*
- *Which of the concurrency abstractions we have discussed till now can be implemented using one-slot-buffers as the only synchronized objects?*
- *When are futures better than early replies? Vice versa?*