

Design of the p2p simulator

Juanjo Aparicio Jara

Design of the p2p simulator

Juanjo Aparicio Jara
Copyright © 2005 The DBE Project

Table of Contents

Preface	v
1. Goals	1
Description of the problem	1
User stories	2
2. Approach to modeling and simulation	3
3. Design iterations	4
First iteration	4
Second iteration	4
Alternatives explored in the second iteration	5
Different idea	5
Third iteration	6
Definition of TaskThread, ThreadPool, Scheduler, LockObject	7
A step back	11
Collaborations between the states, the thread, the scheduler and the thread pool	12
Network simulation	15
Lookup phases	15
Definition of lookup phases	16
A general method for thread synchronization	17
Discrete time vs continuous time simulations	18
Continuous Time peer-to-peer network simulator	18
Bibliography	20

Preface

We will discuss the approaches for a successful simulation of common peer-to-peer systems. The design and implementation iterations will be explained in detail, feeding back the results of implementations to a new design phase, that will start the next iteration.

Chapter 1. Goals

The goals of this project are:

- To provide a framework for the testing of mechanisms and policies for *peer-to-peer* networks.
- To enhance the behaviour of *FADA* [<http://fada.sourceforge.net>] by studying of different mechanisms and policies through the use of the simulator.

Description of the problem

The simulator will have a pool of worker threads. Whenever a request is performed on some node, the scheduler will assign a worker thread to that request. When the request finishes the worker thread goes back to the pool. However, only one thread is awake at any moment in time. In that way, the 3000 threads limitation affects the maximum number of concurrent requests. The number of nodes is only bounded by the memory available to the JVM, which is only bounded by the total system memory (physical + swap area, if it exists).

The simulator will keep a queue of threads to be scheduled. Following the definition of concurrency given in 2, two processes are concurrent while there are no causality relationships between them, that is, they don't interact. When they interact, each of them already know what happened first: the sending of the interaction message precedes causally the reception of the interaction message. In this way, it is possible to determine the order of execution of concurrent processes: if they don't interact the order is irrelevant; if they interact, the order is specified by the causality principle: causes precede results. Likewise, the simulator scheduler will schedule the threads so their causality order is preserved.

From this explanation follows that the interesting entities to model are:

- *The scheduler*. Selects a thread from the scheduler queue to be run. Also keeps track of the passing of logical time, although this role could be implemented by a global clock entity (class).
- *The scheduler queue*. Keeps the list of threads to be run. Actually, there will be two queues, one for active threads and one for blocked threads. Every scheduler has one or more scheduler queues.
- *Threads*. Execute processes that serve requests. When a thread synchronizes with another thread it will do so by blocking on a lock object. Threads can block also for a specified amount of time. Threads are stored in scheduler queues, and are obtained from a thread pool and returned onto it when they have finished.
- *Thread pool*. Stores all created threads. The threads in the pool are created at the beginning of the application run, and threads are given when they are needed, and restored to the pool when they finish. Actually this serves the purpose of enhancing the performance of the simulation, so it is a disposable class. However, with a view in future requirements, providing a dummy thread pool that creates threads on demand and destroys them when they are released would provide a simple, non-performant implementation of the thread pool interface.
- *Lock objects*. These are the synchronization points among threads. They are controlled by the scheduler, so when a thread block on a lock object the scheduler will put the running thread onto the blocked processes list
- *Requests*. Generated by the scheduler by following a *random distribution*.
- *Nodes*. Obvious item

- *Mechanisms.* The algorithms that implement several operations in a node: lookup and register, mainly. Other operations supported in FADA are the connect mechanism, usually triggered by the lookup operation, and disconnect mechanism, usually triggered by a human operator.

Processes (requests) may also spawn new subrequests (such as the ones present in the flooding algorithm.)

User stories

A request is assigned a thread to work. The thread calls the request method on the node, which delegates to the proper mechanism. The mechanism executes in full (using the same thread), then finishes and returns the results. The thread becomes available for another request.

In the lookup mechanism the thread creates several other threads, one for each neighbor, that in turn execute the extendLookup mechanism. The main thread does not join those threads, but waits for a timeout to happen, or for the number of desired responses to be reached, whatever happens first. The main thread blocks onto a lock object that is accessed by other requests. The scheduler must check lock objects when they are accessed and reschedule blocked threads. They return momentarily to running state, then they maybe return to blocked state. **wait()** and **notify()** can be overwritten (hopefully), but **synchronized** blocks are more difficult to synchronized with the scheduler.

Chapter 2. Approach to modeling and simulation

The simulation framework should allow the user to create classes that encapsulate the different policies and mechanisms under test. Ideally, those classes could be plugged in a real system. However, this is a secondary goal, because the execution environment of a simulation is quite different from the execution environment of a real system.

In a real system, the application is free to create as much threads as needed by following any of the existing thread management patterns. In a simulation, though, the framework must simulate hundreds, thousands of nodes. The memory and CPU requirements scale linearly with the size of the simulated network. For this reason, the execution environment of the simulator will be constrained to allow the simulation of thousands of nodes without running out of system resources.

One approach suggested by Javier Noguera (javier.noguera@techideas.info) is to have each simulated entity (node) to iterate over a sequence of, say, ten steps. The transition between these ten steps is controlled by the simulation scheduler. This allows the suspension of tasks. However, without further explanations, this approach does not seem to fit the problem, because the execution of a task is not a sequential process.

Chapter 3. Design iterations

The design and implementation of the peer-to-peer simulator will follow an iterative approach, because of two factors:

- The peer-to-peer simulator is a high risk task. It is therefore necessary to follow a design and implementation approach that allows the early identification of risks, and a way to reorganize the work to do.
- An iterative approach to design and implementation is better than a pure top-down approach even for lower risk tasks. Iterative design and implementation approaches usually embed the testing phase within the development cycles, leading to better quality software, and the identification of inconsistencies in the requirements specification.

Additionally, the concepts of *literate programming*¹ are an attractive approach to provide implementation and documentation in an as much consistent as possible way.

First iteration

In the first iteration it was attempted to provide as general a simulation framework as possible. That means allowing arbitrary pieces of Java code to be plugged in the simulator to see how the peer-to-peer network reacts.

To be able to execute the simulation steps in an out-of-order fashion a global ordering of the events was needed. The first iteration chose a global clock to provide the global ordering of events, though a decentralized approach is possible².

The total ordering of events allowed to create a thread scheduler. This scheduler takes into account the global clock value, and queues threads onto a priority queue. The scheduler dequeues the head of the queue and sets that thread running, until it stops again. Threads stop by calling the scheduler itself, and notifying it of its intention to stop for a given amount of time. This is not real time, however, but the virtual time given by the global clock.

In order to be able to suspend threads and to resume them afterwards, instead of using the deprecated *stop* and *resume java.lang.Thread* methods, the scheduler suspension method would force the thread to *wait* on a global object.

The maximum number of nodes that can run in a simulation in this way, provided that each node is assigned a thread, is limited by the JVM implementation capabilities, the operating system that hosts the JVM, and the amount of free memory available. Ideally, it should be in the order of the tens of thousands.

However, tests showed that less than 5000 threads in total were created, even though all of them save by one were blocked, and only one of them was running, at any time in the simulation

Clearly, another method is needed. This is discussed in the following iteration.

Second iteration

¹See [literateProgramming]Donald Knuth's page [<http://www-cs-faculty.stanford.edu/~knuth/lp.html>] on the subject

²See Lamport78 [<http://research.microsoft.com/users/lamport/pubs/time-clocks.pdf>]

The approach of the first iteration tried to provide complete transparency to the programmer, so the algorithms could be unplugged from the simulator and plugged into real peer-to-peer software implementations. The increasing number of problems to achieve that goal makes us perceive a smell that indicates this is not the way to go. An alternative approach is needed.

Stepping back a little, taking a concrete example, we can have a look at FADA.

Each FADA node runs in its own instance of the JVM. When no requests are made to a FADA node, there is a thread running continuously that checks the expiration times of registered items (if any) and the passing of time in order to delete items that are out of date. In absence of registered items, the thread remains blocked (consumes no CPU cycles).

There is another thread that is blocked onto a `socket.accept()` operation. This thread accepts requests from the outside of the node. Whenever a client or FADA node sends a request to this node the thread is awoken, and a socket is given to it. The thread then spawns a new thread to fulfill the request, and blocks again on the `socket.accept()` call.

When a request is made to a FADA node, such as a `lookup()` request, the thread that fulfills the request creates more threads, one to perform a lookup on the local item registry, and another one for each of the known neighbors. The latter threads will call remote methods on neighbors. These calls are performed in an asynchronous fashion, that is, the caller does not wait for the callee to perform its duties, but returns right after performing the call.

After speaking with the boss, we decided to move to a simpler model, and leave the full modeling for a later iteration.

Alternatives explored in the second iteration

JPDA

The Java Platform Debugger Architecture was an interesting option to investigate. It provides a framework for debugging Java classes, which allows the suspension and resumption of the JVM execution at arbitrary points. However, the thread restrictions that were valid for the multithreaded approaches also apply here: threads can not be reassigned to different classes during runtime.

Multiple machines

One way to deal with the limitation imposed by the JVM implementation regarding the maximum number of threads to execute is to have several JVM instances, in the same or different machines, to perform the simulation. This approach has the obvious drawback that the simulator is a p2p application itself, so all the burden of coordination in decentralized system is applied to an inherently coordinated process as is a simulation. This is not to say that distributed simulations are not a way to go, but rather that they are better suited to problems where a loose coordination is in place³, or where coordination efforts are not the hard problem to solve. The simulator problem is, first and foremost, a problem of coordination.

Different idea

I4 think I've been following the wrong approach. In a real system designed following the OOP principles, each of the interesting problem entities is modelled as a class or class hierarchy. Active entities (as defined in Booch's "Object Oriented Design with Applications") receive one or more threads to perform their duties. But in a simulation the modeling differs from the actual application modeling. In a

³See the `seti@home` [<http://setiathome.ssl.berkeley.edu/>] homepage

⁴Juanjo Aparicio [<mailto:juanjo.aparicio@techideas.info>]

simulation, the main abstractions should be events, processes and the scheduler (as was done in the section called “First iteration”). In that simulation model, each of the nodes in the p2p network received a thread, that is, became an active object, mimicking the behaviour of the real system. However, the limitation of 2000-3000 threads per JVM posed a constraint on the maximum number of nodes in the simulations. Following the alternative approach, we have a limitation of 2000-3000 *simultaneous requests*, while the maximum number of nodes is limited by the memory available to the JVM, which is parameterized.

Third iteration

We start by redefining the full FADA model, and go for a simpler one, which can (hopefully) be modeled in an easier way.

Let's consider the lookup operation as is actually implemented in FADA 5.2.6. A client makes a lookup request to a FADA node. The FADA node traverses the list of neighbors, and spawns a task for each of them. When all tasks have been created and are running, the FADA node attempts to match the given search template against the local registry. All matches are added to a result container. Then the FADA node sleeps for the amount of time specified in the lookup request. Meanwhile, the spawned tasks call the `extendLookup` method on the neighbors of the node.

During the time the FADA node is performing the local lookup and then sleeping, other nodes of the network may have found matches for the search template, and they contact the original FADA node and notify it of these results. The results are added to the result container for the current search, which has an identifier to distinguish it from other result containers for concurrent searches. When the thread awakes, it returns all achieved results to the calling client.

This approach attempts to provide the smallest return time, but it makes heavy use of existing resources (connections to other nodes, threads, ...). It has the additional problem that the original node doesn't know when the search has finished, so if the whole network is traversed before the timeout is reached, the FADA node will sit idle until it times out.

An alternative approach, less resource-consuming, would be as follows: instead of making calls to the `extendLookup` method of all the neighbors, and then expect asynchronous requests that contain responses achieved by remote nodes, the node that calls `extendLookup` may block until the called neighbors respond. When they do, the original FADA node knows the lookup has finished, and doesn't need to wait any additional time.

This approach has differences with the original algorithms. First of all, the node can not return results when it reaches the timeout, because the other nodes may not have responded yet. Second, the original node does not know which nodes provided the answer, because all it sees are responses from its immediate neighbors, and thus it can not connect with unknown nodes.

To alleviate the first condition, it may be possible for the nodes to start returning results as soon as they have them, but don't declare the operation as finished until the nodes it has contacted declare it finished. An operation is declared finished when all neighbors declare it finished and there are no more neighbors that are willing to accept the same lookup request. However, using an RPC-style of calls, it is not possible to have two responses separated in time.

To alleviate the second condition, each result can include the identifier of the node that obtained the match by looking at the local registry. No additional modifications are needed for this case.

In this alternative approach, the call to `extendLookup` may be performed in parallel or serialized in time.

This way of modelling processes doesn't allow tasks to block on other tasks in an asynchronous way (as achievable with the `wait()` and `notify()` methods of the `java.lang.Object` class), for the reasons depicted in the section called “First iteration”. It is easier to implement, but it is not *yet* clear if it is accurate enough.

Definition of TaskThread, ThreadPool, Scheduler, Lock-Object

TaskThread

A TaskThread is a subclass of thread that is aware of the presence of the ThreadPool and the Scheduler. This awareness does not reflect on its public API, but rather in its private interface and as implementation details.

A TaskThread is created with an instance of Runnable, which may optionally be empty (null). Therefore, a no-argument constructor would be possible. Before a TaskThread is run an instance of Runnable must be set to it, either via the constructor or via the setRunnable() method. When a TaskThread is created its state is NEW. When a valid (i.e. non-null) runnable is set the thread state is RUNNABLE.

A TaskThread is started by calling its start() method. When this method is started its state switches to Thread.State.RUNNING. The TaskThread enters the runnable queue of the scheduler.

When a TaskThread locks on a LockObject the TaskThread is assigned to the waitSet of the lockObject. It will remain there until the TaskThread is unlocked. More on this in the section about LockObject.

When a TaskThread finishes running the Runnable instance its state becomes TERMINATED, and it can be recycled. This is so the performance of the simulation can be shifted up a bit. Collaboration with the ThreadPool is needed.

When the Scheduler starts a TaskThread it doesn't know if it is brand new or is a recycled one. It could be possible for the scheduler to find out, but a simpler way would be for the TaskThreads to return to the same state they are when they are created new, or for new threads to go to the state they are when they are recycled. Given that a TaskThread may be recycled many times, this is the general case, so the implementation should be optimized for this case.

Given that when a thread is NEW it is in the state new, and when it finishes it is in the state TERMINATED, the question is what to do with threads that finish and are recycled. When they finish they become TERMINATED. When they are reassigned they become NEW. A thread in either state will not be scheduled: only threads that are RUNNABLE may be scheduled.

To reassign a thread (make it go to the NEW state) the reset() method could be use. This method would have a package visibility modifier (default modifier). A simpler option is to allow threads that have finished execution to stay in the TERMINATED state, and when the setRunnable() method is called with a non-null runnable the thread becomes RUNNABLE.

The Scheduler will only allow a single instance of the TaskThread to run concurrently. This is inherently less efficient than having more than one instance of TaskThread running, but considerations on this advanced behaviour will be deferred to a later iteration. The TaskThread that is running will run indefinitely until it attempts to acquire the monitor on a LockObject. In that moment the Thread will become BLOCKED, and the TaskThread state can't be changed. This state is not reflected in the TaskThread class, but it is implicit in the Thread behaviour.

When the Thread acquires the monitor it will continue running. If it ever calls the wait() method on a LockObject (or a similar method, as the wait() method is a final method) it will become part of the wait-Set of the LockObject, and will abandon the runnable queue of the Scheduler. The Scheduler then selects another TaskThread to run. The blocked TaskThread will enter the state Thread.State.BLOCKED and the Scheduler will be notified of this condition. This is done before the real wait() method is called.

An interesting problem is the state the TaskThread is when the method run() finishes. If the TaskThread becomes Thread.State.TERMINATED, the waitToFinish() method only has to test that condition, but the thread can not be rerun. In order for the thread to be rerun the run() method must not be abandoned. If the run() method is not abandoned, the TaskThread must become Thread.State.NEW again. For that,

the `waitToFinish()` method must check that the state is either `TERMINATED` or `NEW`. Alternatively, different states can be used (instead of sticking to `Thread.State`). One possible solution is to have internal states different to `Thread.State`, but the method `getState()` translates these internal states to `Thread.State`. The internal states are the same as `Thread.State`, but add the state `KILLED`, for when the `run()` method is abandoned. This solution adds little value to the `TaskThread` implementation, and only save a comparison in the `waitToFinish()` method.

State switching

The change of state in a thread must be performed in a way that keeps consistency between all threads that may call methods on the thread that is changing the state. To provide that, a thread's state object is in reality a wrapper around the state objects that follow the State design pattern. The class `StateImpl` implements the interface `State`. The interface `InnerState` extends the interface `State`. The classes `NewState`, `RunnableState`, `RunningState` and `TerminatedState` follow the State pattern and implement the `InnerState` interface. The `StateImpl` class has an instance of `InnerState`. Most methods provided by the `State` interface are delegated by `StateImpl` to the actual `InnerState` implementation. In this way a general interface is provided for interoperability reasons, and the behaviour is mandated by each implementation of the `InnerState` interface.

Each implementation of the `InnerState` class represent a state in the `TaskThread` model that is being described in this section. Each of these classes has a `getState()` method that returns an instance of `java.lang.Thread.State`. There are more implementations of `InnerState` than different instances of `java.lang.Thread.State`. In particular, both the `RunnableState` and the `RunningState` return `java.lang.Thread.State.RUNNABLE` when its `getState()` method is called. This is so because the `java.lang.Thread` interface doesn't make the distinction between a thread that can be run and a thread that is effectively running, but has not been started yet. However, in the implementation of the `TaskThread` this distinction is important.

The general mechanism for changing a thread's state is to acquire the monitor on the thread's State object, change its `InnerState`, and notify all threads that are waiting on the State object (*not the InnerState*.)

Let's clarify it with an example. A thread `t0` is created in the state `NEW`. The thread is not running, and its `runnable` is not set (it is null). An external thread `t1` calls the thread `t0`'s `setRunnable` method. The `setRunnable` method delegates the call on `State.setRunnable`, which is really executed by `StateImpl.setRunnable`. `StateImpl.setRunnable` delegates on `InnerState.setRunnable`, which, in `t0`'s state, corresponds with `NewState.setRunnable`. `NewState.setRunnable` calls `t0`'s `setRunnablePrivate` protected method, which sets the `runnable`, then changes the state. To do so, it acquires the monitor on `t0`'s `StateImpl` instance and changes the `InnerState`. `t0`'s main loop was running `NewState.run()`. `NewState.run()` had acquired the monitor on the `StateImpl` instance (not the `InnerState` instance), and was `wait()`ing on it. `NewState.setRunnable()` calls `notifyAll()` on this monitor, and leaves the monitor. `t0`'s main loop thread awakes, and the execution of `NewState.run()` continues. `NewState.run()` reaches the end of the method and returns. `t0`'s main loop executes again, running the `StateImpl.run()` method. This method, again, delegates on `InnerState.run()` method. But this time `InnerState` is `RunnableState`, therefore `RunnableState.run()` method begins execution. In the meantime, what happened to `t1`'s thread that initiated the process? When the call to `NewState.setRunnable` finishes, `t1` continues execution, unaffected by the changes in `t0`.

There may be other cases where the transition between states is not as easy, especially transitions that involve states whose `run()` method is somewhat complex. Let's check this out with an example:

Suppose thread `t0` is in state `RunningState`. The `RunningState.run()` method must execute `t0`'s `runnable run()` method. When that method finishes the thread must go to the state `TERMINATED`, represented by `TerminatedState`. The `RunningState.run()` method finishes by acquiring the monitor on `t0`'s state object, changing the state to `TerminatedState` and releasing the monitor. No notifications are needed. The `TerminatedState.run()` method will start when the thread's main loop loops again.

ThreadPool

It is the area that keeps instances of `TaskThread`, ready to be assigned an implementation of `Runnable`. Threads that finish their execution return to the pool to be assigned again.

Keep in mind that there is a problem. When a `TaskThread` finishes, it returns to the pool, but the client still has a reference to it. This can be fixed by interposing an additional indirection level, that is, making `TaskThread` an proxy class that delegates to a real `TaskThread`, and when the `TaskThread` finishes, it returns the real `TaskThread` to the pool and throws an `IllegalStateException` when any of its methods are called on the now invalid instance.

Scheduler

The Scheduler is the virtual processor that coordinates the order in which the `TaskThreads` run. The running of `TaskThreads` is continuous, undisturbed by the scheduler, save by when a `TaskThread` wants to sleep for a specified amount of time, by calling the static method `TaskThread.sleep()`, or the `TaskThread` attempts to acquire a lock on a `LockObject`, or a `TaskThread` that has acquired such a lock calls the `wait()` method on it, either with a long parameter or without it.

When a `TaskThread` calls the method `sleep()`, the scheduler takes the `TaskThread` to the state `TIMED_WAITING`, and puts it in the scheduler queue, with a wakeup time that equals the actual time the moment the `TaskThread` called the `sleep` method plus the amount of time specified as a parameter. When the logical clock advances to that time, the scheduler puts the `TaskThread` in the `RUNNING` state and allows it to run again.

When a `TaskThread` tries to acquire the lock, it becomes `BLOCKED`. The `TaskThread` is deleted from the scheduler queue and enters the queue of threads that are competing to acquire the monitor on the `LockObject`. When one of these threads acquires the lock it becomes `RUNNING`.

When a `TaskThread` acquires a lock and then calls the `wait()` method, it becomes `WAITING`. The `TaskThread` is deleted from the scheduler queue and enters the queue of threads that are waiting for the monitor on the `LockObject`. When one of these threads is notified it acquires the lock and becomes `RUNNING`, returning to the scheduler queue. When all of these threads are notified, they compete in the usual way (see JLS doc or JVMspec doc to know the details of competing), so only one of them becomes `RUNNING`, and the rest become `BLOCKED`.

When a `TaskThread` acquires a lock and then calls the `wait()` method with a long parameter, it becomes `TIMED_WAITING`. If the timeout goes by, it becomes `RUNNING` and competes again in the usual manner to acquire the monitor on the `LockObject`.

When a `TaskThread` releases a `LockObject` it becomes eligible for scheduling, and goes back to the scheduler queue. The Scheduler will pick up a Thread in the queue and resume its operation. The Scheduler should *not* need to know what state the thread is: if suspended, brand new or recycled. That means that the same method should be used to continue execution of a thread in either of the three conditions. The method can be called `start()`, but then the `start()` method is overloaded, its semantics altered, and harder to understand if the `TaskThreads` are used in standalone, without a scheduler. A `continue()` method seems a better alternative, but that also means that the threads must be started by default when they are used in conjunction with a scheduler, so the `continue()` method is meaningful for them. Alternatively, the `continue()` method could either resume or cold start a `TaskThread`. This alternative is easy to understand and provides a cleaner interface, although the semantics of the method depend on the context in which it is used. However, its effects are the same in any context: the thread starts running.

In other words, when a thread is created by a standalone application, no scheduler will be set. When its `start()` method is called, the thread should start execution right away. When a thread is associated with a scheduler, its `start()` method sets it up so it can start whenever the `resume()` method is called. When the thread blocks, the `resume()` method allows the execution of the thread to continue.

LockObject

A `LockObject` is an object which will be used solely for synchronization purposes. When a `TaskThread` wishes to access some variable in mutual exclusion mode it will do so by acquiring the lock on the `LockObject`. When it leaves the mutual exclusion zone, it must free the `LockObject`. To avoid deadlocks, it is advised to keep the mutexes as small and controlled as possible.

When a `TaskThread` wants to acquire the monitor on a `LockObject` it does so by calling the method `acquireMonitor()`. Multiple `TaskThreads` attempting to acquire the monitor on a `LockObject` compete in the same terms as instances of the class `java.lang.Thread` when acquiring the monitor of a `java.lang.Object`.

When the method `acquireMonitor()` is called by a `TaskThread`, the `LockObject` calls the `Scheduler` and instructs it to delete the `TaskThread` from the running queue; then the `LockObject` puts the `TaskThread` on the competing queue. The time at which the attempt to acquire the monitor was performed is annotated in the competing queue. The `LockObject` then checks to see if there's already a `Thread` that owns the monitor. If it is, it does nothing: the `TaskThread` remains in the `BLOCKED` state.

The monitor primitive needs assistance from the compiler. In the Java language, this is accomplished by making monitors *part* of the language, so the compiler/interpreter treats them as first class citizens. Our classes are not distinguished with such honor, therefore we can not do things like enclose a critical region in a block for automatic acquirement and release of the monitor. Therefore, we can switch our view to a primitive that requires no assistance from the compiler/interpreter: a semaphore or a mailbox.

Semaphore

A semaphore is a primitive first proposed by Edsger W. Dijkstra in 1968. It has two operations: `down()`, to enter a critical region, and `up()`, to leave it. A semaphore has an integer variable set to some value. This variable, if positive, indicates how many process can enter the critical region at once.

`down()`: When a process wishes to enter a critical region it first performs a `down` on the semaphore. The integer variable is decremented. If the integer variable is greater than or equal to zero the process continues; otherwise the process blocks.

`up()`: When a process within the critical region leaves it it must explicitly call an `up` on the semaphore. The call on `up` causes the integer variable to be incremented. If the variable is negative this means that there are process blocked, waiting to enter the critical region. The semaphore then wakes one of these processes.

Both of these operations are atomic and mutually exclusive: a thread that enters the `down()` operation forbids any other thread to enter the `down()` or `up` operations(). They will be blocked until the first thread finishes, either by blocking or by continuing execution. Then the other threads may enter the `down()` and `up()` operations.

Note

The usual way to call the `down()` and `up()` operations is `wait()` and `signal()`, respectively. However, to avoid confusion with the `wait()` operation already present in every instance of `java.lang.Object`, we have chosen to use the somehow more obscure names `down()` and `up()`. Hopefully, readers familiar with semaphores will not find it that difficult.

The semaphore primitive has the drawback that it must explicitly invoked. If a process forgets or fails to call it properly, this can lead to deadlocks and race conditions. Monitors need no explicit code to block and unblock processes, because the primitive is embedded in the compiler. On the other hand, if you don't have complete control over the compiler or the execution environment it is very difficult to implement a monitor. In this case, a semaphore can be used to provide mutual exclusion in the access to a critical region.

Mailbox

The mailbox is another primitive for process/thread synchronization. The technique is also called message passing.

A process that wants to synchronize with another process performs a `receive()` operation on a mailbox. The process blocks until a message is received. The other process performs a `send()` operation on the mailbox when it wants to wake up the other process. In that moment, both processes continue running.

Messages are queued in the mailbox, so if multiple messages are sent to a mailbox before any of them is received, they are kept in a buffer for the consumer to receive them.

The obvious drawback is that one process takes the role of sending messages, while the other takes the role of consuming them. This is perfect for a producer-consumer style of problem, but performing mutual exclusion on a critical region when the order of execution of the processes is not known takes a little bit of thinking.

A step back

Looking closely at our problem, the synchronization needs are limited to:

- Avoid concurrent access to data objects
- Allow threads to wait for conditions to happen. For instance, a lookup thread will wait for responses to be collected. Timed waits must be allowed

To avoid concurrent access to data objects we will allow the programmer of the simulation freedom to use the usual synchronized Java primitives. The programmer is advised to keep such synchronized blocks as small as possible.

To allow threads for conditions to happen, the `LockObject` will provide the following operations:

- `block()`: The calling thread will block until the thread is awoken externally (through `release()` or `releaseAll()`).
- `block(long milliseconds)`: The calling thread will block until the specified number of milliseconds have passed, or the thread is awoken externally (through `release()` or `releaseAll()`).
- `release()`: The calling thread will awake one of the threads blocked on this condition.
- `releaseAll()`: The calling thread will awake all of the threads blocked on this condition. The blocking threads will then compete for the lock in the usual manner.

This closely mimics the behaviour of the `java.lang.Object` `wait()`, `notify()` and `notifyAll()` methods. The main distinction is that the acquisition of the object's monitor is implicit in these methods, and that the scheduler is aware of them, while the mentioned methods will suspend and resume the execution of threads without the scheduler noticing. The intention is that the scheduler is able to schedule threads by their logical time, of which the JVM scheduler is not aware.

Remember one thing: in order to call the monitor methods on an object it is not necessary to embed the call in a synchronized block. It is enough that the caller has done so. So it is possible to have `block()`, `release()` and `releaseAll()` methods that are not synchronized to the object, and the caller **MUST** enclose them in synchronized statements. In this way we use the mutual exclusion abilities of FADA, while being able to lock and unlock on the monitor, and schedule the thread in and out of the cpu.

After doing some tests with `BlockingPoint`, and adding the methods `pause()` and `release()` to `TaskThread`,

I realized that when a Thread acquires a lock on Object a, then acquires lock on Object b, then wait(s) on Object b, the monitor on Object a is not released. Thus, if BlockingPoint.block() acquires the monitor on the instance of BlockingPoint, then calls TaskThread.pause(), which acquires the monitor on the instance of TaskThread, and waits for another thread to call TaskThread.release(), via BlockingPoint.release(), the second thread can never call BlockingPoint.release(), because the method is synchronized, and the monitor is owned by a thread that is waiting on another monitor. Either we make BlockingPoint.release() not synchronized, or we forget about synchronizing on two monitors at once.

Collaborations between the states, the thread, the scheduler and the thread pool

The main goal of the thread is to run a task, then go to a state where it can be recycled. The main goal of the thread pool is to create threads on demand and giving them to clients. When threads finish, they notify the threadpool so they can be recycled (assigned again to an asking client). The main goal of the scheduler is to select a thread to run. The selected thread runs until it finishes. When the thread finishes it notifies the scheduler. The scheduler then removes the thread from the queue of running threads and selects another thread to be run.

The thread states are modelled following the State Pattern. Each of the thread states is given its own class. All of these classes derive from a superclass. The run() method on the thread does the following: it waits until a notify is called on the state. When this happens, it calls the state run() method. Then it waits again. This is an endless loop, only broken when the thread is interrupted, as a result to a call to halt()/kill().

There are four different states, named NewState, RunnableState, RunningState and TerminatedState. They all descend from the superclass InnerState. The class InnerState defines the following methods:

- start(): This is called initially by an entity external to the thread. The semantics of the method is to make the thread start running.
- run(): This is called by the thread inner loop. When a thread starts running, the runnable will wait on the state; when the wait is released, the runnable will call the run() method on the state. When the run() method on the state is finished, the loop closes and the runnable waits again.
- halt()/kill(): Called when it is desired to stop a thread which is running. A thread that is stopped may not be restarted again. It should disappear from the threadpool and the scheduler.
- joinThread(): The calling thread will block until the target thread finishes its execution, either by arriving at the end of the runnable method, and therefore reaching the Terminated state, or by being interrupted by means of the kill()/halt() methods. In the latter case all threads that are joining this thread should be notified.
- setRunnable(): This method belongs to the private interface of the Thread class. It is used when a thread is recycled, to set the new instance of Runnable to be run. The behaviour is similar to what worker threads do.
- setState(): Called only by other states. This method changes the state in which the thread is. After changing the state, the state instance is notified, to awake the inner loop of the thread class.
- getInnerState(): Used by other states to know what is the state of the thread. Ideally this method should be redundant, as the thread is only in one state, and all the states are executed by the same thread.

The usual lifecycle of a thread is as follows:

1. The thread's class constructor is called.
2. The thread state is set to New
3. The thread's setRunnable method is called.
4. The thread's runnable is set, state changes to Runnable
5. The thread's start method is called
6. The thread's state changes to Running
7. The run method on the runnable finishes, state changes to Terminated
8. The Terminated state calls the scheduler and the thread pool, notifying the new state.

The meaning of each method for each of the states is as follows:

NewState

All threads are in this state when the thread constructor finishes. In this state it is only legal to set the runnable, kill/halt the thread and join it. The rest of operations are either illegal or have no effect. A new thread starts in this state, with its run() method executing NewState.run() method. When another thread calls the thread's setRunnable() method, the thread will delegate the call to the state setRunnable(), in this case NewState.setRunnable() method. This method will change the state to the RunnableState. The change of state is done in a synchronized way, notifying all threads when the change is performed. This is meaningful because the thread's own thread will be executing NewState.run(), which blocks waiting on the state object. Then another (external) thread will call the setRunnable method, causing the NewState.run() method to awake, see that it is no longer the current state and finish. The thread's thread run() method loops and executes the new state run() method. The new state will be RunnableState. Calls on the thread's public methods that affect the state of the thread must wait when the state is changing, otherwise the notifications will be lost. As the change of state is only performed by the State classes themselves, they must wait on each other before they go on.

- setRunnable(): Executed by an external thread. Calls setRunnablePrivate on the thread, switches the state to RunnableState, finishes. This causes the thread main loop to execute the RunnableState run method
- start(): Executed by an external thread. A thread can not be started until its runnable instance is set. Therefore, this method throws an IllegalStateException.
- run(): Executed by the thread's main loop. This method blocks on the thread's state. When it is awoken it checks to see if the state is still NewState, and if so, it blocks again. If not, the run() method finishes. This causes the thread's main loop to execute the new state's run() method.
- halt()/kill(): Executed by an external thread. The state is switched to TerminatedState, and the method finishes. This causes the thread's main loop to execute the TerminatedState.run() method.
- joinThread(): Executed by an external thread. Returns immediately, as the semantics of joinThread is to wait for a thread while it is running. A thread in the NewState does not run, therefore the thread can be joined immediately
- getState(): Executed by an external thread. Returns the java.lang.Thread.State instance that matches this state. In this case, it is java.lang.Thread.State.NEW

RunnableState

When a thread's runnable has been set, it can be started. Therefore, it goes to a state where it can be run, or a runnable state. This state represents that condition. In this state, the runnable may be changed again. It is in this state, also, where the start() method becomes meaningful. When the start() method is called on the thread, it will delegate onto this state, that will switch the state to running.

- setRunnable(): Executed by an external thread. Does the same as the NewState.setRunnable method. It is legal, because the thread has not yet been started.
- start(): Executed by an external thread. Switches the state to RunningState.
- run(): Executed by the thread's inner run() method. Blocks while the state is RunnableState. When the state is no longer RunnableState it finishes. The thread's main loop will execute then the run() method on the following state.
- halt()/kill(). Execute by an external thread. Switches the state to TerminatedState.
- joinThread(): Executed by an external thread. While the thread's state is Runnable, this method waits on the state. Then executes the new state's joinThread() method.
- getState(): Executed by an external thread. Returns the java.lang.Thread.State instance that is associated with this state, java.lang.Thread.State.RUNNABLE in this case.
- getInnerState(): Executed by an external thread. Returns this instance of com.sun.dbe.p2psim.InnerState.

RunningState

When a thread has been started and it's executing its runnable run() method, the thread is in this state.

- setRunnable(): This method throws an IllegalStateException, as the runnable can not be changed while the thread is running.
- start(): This method is ignored, as the thread is already running.
- run(): Executes the thread's runnable.run() method, then switches the state to TerminatedState.
- halt()/kill(): Interrupts the thread's main loop (via a call to interrupt()). The thread is no longer restartable. This is due to the impossibility to tell the user's runnable that it must exit the run() method, and... why not make the user implement an extension of the java.lang.Runnable method that *has* a method to notify the intention to halt the thread?
- joinThread(): Acquires the monitor on the thread's StateImpl object and checks to see if the state is Terminated. If not, the current thread calls wait() on the state. When it is notified, it checks the state again. A while() construct is used for that.
- getState()
- getInnerState()

TerminatedState

The TerminatedState represents the end of the thread lifecycle. When a thread finishes the runnable

run() method, it will go to this state. In this state, the thread will wait for the setRunnable method to be called, which will take the thread to the Runnable state again. The rest of methods will do little or nothing at all.

- setRunnable(): Does the same as NewState.setRunnable().
- start(): Throws IllegalStateException. A terminated thread can't do anything but go to the Runnable state.
- run(). Loops waiting on the thread's state until the thread's state becomes RunnableState.
- halt()/kill(). Returns immediately, as if the thread had been halted/killed right now.
- joinThread(): Returns right now. The method joinThread waits for the thread to die, and the thread is already dead.
- getState()
- getInnerState()

Network simulation

The simulator, at least in its early stages, will only simulate <n> operations

Find a node: Use the bootstrap mechanism to find a node in the network.

Register: Once a node has been found, register some data in that node.

Lookup: Once a node has been found, start a search for something on the whole network, starting from that node. This operation is arguably the most complex operation in FADA, and the simulator must be able to accurately mimic the behaviour of FADA as it is, and to be the testbed for more advanced algorithms to be used in FADA.

Lookup phases

When a node receives a request for a lookup, it generates a locally unique search identifier. It then searches the local registry. After the search in the local registry has finished, or in the meantime if a threaded approach is used (the simulator must support both), the node selects a non-empty subset of its neighbors (the dumbest approach is to select them all) and extends the search to those nodes. These nodes will receive the search criteria and the search id as well. When a node is asked to perform a search whose search id has been already seen by the node, the node declines, because it has already performed that search in the past. Ideally, the seen search ids should be stored in a structure that deletes them after a certain amount of time has passed.

When a node has finished searching all of its neighbors and has finished searching its local registry, it returns the results achieved so far to the calling node. The calling node then merges the results achieved so far with the results just received. This merging operation gets rid of duplicate results.

In FADA, a node expects no response from its neighbors, but enters a state where it is waiting for a timeout to go by, or the asked number of responses to be achieved. However, such behaviour is difficult to model with a single threaded simulation. In the next iteration, the simulation code will be mixed with the thread, pool and scheduler code from previous iterations, to allow a multithreaded simulation. Note, however, that the number of threads in a JVM seems to be quite limited wrt the number of processes in a given architecture.

From the description above, we can separate mechanisms from policies. Both components should be

highly parametrizable, either by explicit parameters or by pluggable code. The latter approach is more flexible, because it doesn't constrain the versatility of the simulator as much as a fixed (though extendable) set of policies/mechanisms does.

The identified mechanisms and policies are:

- Discovery of the node the search starts with (bootstrap).
- Selection of nodes to which the search will be extended.
- Mechanism by which the search request is propagated: sequential, threaded/simultaneous, node waits for neighbors to finish or goes its own way.
- Rewiring of the p2p network during the lookup. In the FADA implementation this operation takes place when a node returns fruitful results. A more general solution should allow this operation to happen at arbitrary places. A middleground solution could be to define phases as in the Apache request servicing, and to allow the user to interpose her code for the different phases. This idea deserves further exploration, IMHO.

Definition of lookup phases

In the section called “Lookup phases” the lookup operation was broken down into several phases. As a way to allow users the greatest flexibility in their simulations, it was proposed to fix the number of phases a lookup request goes through, Apache-request-processing-style, and provide mechanisms for the user to embed her code inside these points.

The bootstrap phase is already parameterized via the `BootstrapMechanism` class.

The selection of nodes is not yet parameterized, but the `Node` class already has a place for it. (*Node now uses a class named `LookupExtensionMechanism`; factory methods for this class are still needed*)

The expansion of the lookup request is performed by the class `LookupExtensionMechanism`. This class has a method called `getNeighborsToContact`, which selects a subset of all the neighbors of the node. However, the way in which these neighbors are contacted is not specified. There are various options:

- Extend the search parameters to all the selected neighbors in sequential order, waiting for each neighbor to finish its job before extending to the next neighbor.
- Extend the search parameters to all the neighbors simultaneously, by using threads for the extension. This method requires the lookup request to stop after the extension has been done, blocking the thread. The spawned threads must notify the blocked thread so it may go on. The use of a timeout is highly encouraged.

This mechanism should also be externalized, and facilities must be provided to the programmer of the mechanism. Note that thread synchronization may be difficult to achieve, especially in the presence of remote connections.

A plausible solution to the synchronization of threads in an ordered manner that allows the scheduler to take part in the process could be to have synchronization objects with a `lock()` method that is only released after a timeout or when the conditions apply. The conditions are checked by a programmable class (interface to be defined) with a `check()` method. The synchronization object also has an `awake()` (or similarly name) method that calls another class (interface to be defined) to let it know what happens. In this way there is interaction between the providers of events (those who attempt to awake the thread) and the consumer of the events (the thread that is blocked). Indeed this is a new class hierarchy

that can be extracted out of the simulator, as its benefits as a general synchronization method are pretty obvious. Forward to the section called “A general method for thread synchronization” for more information.

The mechanism by which the search request is propagated is not yet parameterized, and the code does not make provision for this parameterizable code to be applied. Some mechanism is needed. Ideally, this mechanism should be settable either by node or by request. Both approaches have their pros and cons. Parameterizing by node allows for easy deployment: you set the parameterized class at node installation, then the nodes use the same mechanism. However, different nodes with different mechanisms could lead to inconsistent behaviour across the network. The simulator could be used to study the impact of such state of things. Parameterizing by request would make sure the same mechanism is consistently used for the whole request, but it also implies there is a way for the requester to specify such mechanism, and for nodes to propagate such mechanism while searching the network. Java serialized classes could be used for this, but the requester should be able to provide the needed classes to the nodes when they need them. This approach is something I would frown upon. Therefore, the best solution seems to be to parameterize at the node level, and cross fingers expecting the interaction with other nodes will not result in inconsistent results. Within a simulation it is easy to make all nodes use the same mechanism. In the FADA nodes deployed so far this has been assumed.

The mechanism by which the node rewires itself is performed, in FADA, upon receiving the responses from the remote nodes. This is so because it is the moment in which the node becomes aware of the existence of nodes that are not neighbors. One could think of other places where this could be achieved. For example, periodically a node could explore its immediate neighborhood and rewire itself to become more connected. However, by increasing the degree of the nodes the graph will tend to become a complete graph. It is interesting that nodes with a high degree connect to nodes with a low degree. This makes less connected nodes more connected. The degree distribution of a scale-free graph follows a power law, meaning that there are very little nodes with a high degree. A scale-free graph is desirable for its small world properties. Although complete graphs also exhibit small world properties (every node is one hop far from every other node), complete graphs demand more memory per node, because each node has to keep the complete list of nodes of the graph. *(The rewiring mechanism has been externalized now. This paragraph must be updated accordingly.)*

A general method for thread synchronization

Java provides several methods for synchronization of threads. Every instance of a class is a descendant of `java.lang.Object`. That class defines the methods `wait()`, `notify()` and `notifyAll()`. These methods, together with the statement **synchronized** allow code to protect critical regions, block threads that want to enter a critical region and awake blocked threads. The synchronization primitive used is the monitor [<http://sunsite.ee/java/whitepaper/java-whitepaper-9.html>]. While this primitive allows certain degree of flexibility, to support all the monitor features would increase the complexity to the simulator to arbitrary levels (effectively dubbing the functions of an operating system). In order to cut complexity we'll tame down the synchronization needs and define a simple set of synchronization methods, usable within the simulator and hopefully generic enough to allow the implementation of sophisticated peer-to-peer algorithms.

The general technique will be to use synchronization points instead of regions. While this method is less secure, one must keep in mind that what we are trying to model are interactions between peers, and that the interactions between two peers are executed within either of the two. This means that there are not critical regions shared between nodes, because the memory is not shared between remote peers. Therefore we will limit the synchronization object to the checking of evaluation functions, a discrete point in space.

The synchronization object will have one method to lock the running thread, waiting for the condition that releases it to happen. It will also have a method to wake the thread up, so it can check if the condition has been cleared. The synchronization object will have a method to set the instance of a class that checks the condition.

Discrete time vs continuous time simulations

One question that arose when first designing the simulator was if a continuous time simulation offered any benefit over a discrete time simulation. In a discrete time simulation all the operations in the network are synchronized and performed at discrete intervals of time called *steps*. The usual way to work with these steps is that something of interest always happens at each time step. There are no idle steps in the simulation. However, real systems are not usually globally synchronized, and don't pay attention to a global clocking scheme. Parallel processes in different machines run independent of one another. There is no concept of time step. Initially, this seemed like a limitation of discrete time simulations over the real systems, and its effects were not clear to this team. That's why it was chosen to implement a continuous time simulation.

Thinking carefully about it, however, led to the intuition that either in a simulation or a real system, time is only a concept that is useful to determine the causality order of events. That is, in the usual case the absolute value of time in an arbitrary clock is not important. What is important is to know in which order happen the events that have an influence on system state. For example, if a lookup request happens in a given node in the network, and this node propagates the search to its neighbors, and its neighbors propagate the search further, it is clear that the first lookup request preceded the first expansion of the search to the neighbors. If while the request is being propagated, in a different point of the network someone registers something that matches the search criteria before the search request arrives at that point, there is, in the real system, an influence of the latter event over the former. This can be accurately simulated in a discrete time simulation, if the time step is small enough and idle time steps are allowed. For example, the registration in the distant node has to happen before the search request arrives there. The time steps can be made arbitrarily small. In the limit, the time steps are close to 0. In this situation it is impractical for the simulator to run each and every of these states, mainly due to the fact that idle time steps (time steps in which nothing happens) are resulting in a waste of computing resources. A continuous time simulation offers the same results as a discrete time simulation, because the causality order in the latter is the same as in the former. A continuous time simulation can be thought of as a discrete time simulation where the time steps are not necessarily equal, and their duration is governed by the events that happen in the simulation itself. That is, if a lookup request is received in a node and it is processed locally in 10 milliseconds, and nothing happens in those 10 milliseconds, the simulator takes the whole process 10 milliseconds in the future of the simulated time and continues from there.

What is needed in this scenario is a means to ensure that the causality order of the events is preserved in the simulation, in the absence of an auxiliary clock. This has been the subject of the first part of this document, and it's about time that both the thread synchronization primitives dealt with so far and the simulation objects designed in later sections are blended to provide a continuous time peer-to-peer simulation system.

Continuous Time peer-to-peer network simulator

The simulator designed and implemented so far consists of a main class that executes the simulation and calls the various classes that extract statistics about the resulting state of the network. These statistics extractors are pluggable user classes. The various mechanisms and policies are also pluggable user classes. What is lacking is a way to allow these classes to be interrupted at arbitrary instants of time, to be resumed afterwards, and to express dependencies in their order of execution. For example, in the FADA mechanisms as implemented in the FADA release 5.2.6.1, a node that receives a lookup request performs a local search, puts the (possibly empty) results in a lookup results container and spawns threads that extend this search to the node neighbors (that will, in turn, take it further away). The thread that is serving the lookup request made by the client then *waits* for other nodes to notify it of distant search matches. That thread also has a timeout set so that if the timeout is reached before the expected number of responses is reached the results achieved so far are returned to the client, and further results notified by distant nodes are dropped. There must be a way for the simulator to provide this behaviour, and, what's more important, for a programmer to make his user classes work this way without the whole

simulation crashing or blocking due to a deadlock.

Another issue is the limited number of threads that a JVM allows to be spawned. In a single instance of FADA running on a single computer, the exhaustion of threads in the course of the normal operation of the nodes leads to lost requests (because they can not be served) or incomplete searches (because the search can not be expanded to other nodes). In the simulation, though, the effects of thread exhaustion are more dramatic, because a single instance of the simulator has to run hundreds or thousands of simultaneous nodes. While the thread load of the simulation has been decreased by not having the nodes to operate as independent actors, but be mere containers for the policies, mechanisms and their associated data, and having the individual requests use the threads available and span through several nodes, the tree-like structure of the requests imposes the use of several threads for a single lookup request. If the thread pool runs out of free threads some requests will be also incomplete, and the limitations in this case are more severe, because a single JVM has to simulate hundreds of them.

When a request spawns more threads, it must do so by asking the pool (or the scheduler) for one or more threads to complete the request. When one of such threads blocks waiting for external events, it must do so by blocking on the synchronization object described in the section called “A general method for thread synchronization” [17]

With Java monitors the way to work is to have one thread to acquire the monitor, do whatever has to be done in mutex mode, then release the monitor. The monitor may be released in any of two ways: by leaving out the synchronized block, or by waiting on the monitor. Waiting is accomplished by calling the `wait()` method. Either way, there is only at most one active thread in the monitor. The rest of threads, either haven't acquired the monitor, or they are not active. This behaviour can be mimicked in the synchronization object provided by the simulator by having the mutual exclusion zone as narrow as a single method call. There's no need to acquire the monitor: the implementation does so for us. When a node spreads the search over several nodes, it does so by spawning several threads. Then it waits for the timeout to go by, or the number of expected responses to be achieved. This can be modelled with the condition object and the locking object described above. The only thing that must be simulated is the locking of threads: they must go to the scheduler queue, so it can awake them again.

Alternatively, (following the YAGNI principles), we could get rid of the scheduler and have just the pool of threads. This gets more sense once the definition of time above in the section called “Discrete time vs continuous time simulations” [18] has made it clear that we don't need time at all, only proper ordering of events. Instead of one thread per node (as in the original design) or one thread per request (like the second approach), this would lead to one thread per task approach, where the term task is used for the job of serving a user's request or serving a node-to-node request, or whatever job has to be done.

If we simplify by getting rid of the scheduler and keep the lock object, we must also make sure that exhausting the thread pool doesn't lead to inconsistent results. We can cause the `getThread()` method on the pool to be blocking. In that way we are increasing the number of wasted threads, possibly leading to a state where every active thread is waiting for the pool to give it some more threads, and no work is done, so no thread is released, so the active threads stay in the blocked state forever. The funny thing is that the `TaskThread` class will be wasted in this way, as well as all the `InnerState` classes and subclasses. Lesson learned: *YAGNI*.

⁵You aren't gonna need it [<http://www.c2.com/cgi-bin/wiki.pl?YouArentGonnaNeedIt>]

Bibliography

[`literateProgramming`] Dr. Donald Earvin Knuth. DEK. *Literate Programming*
[<http://www-cs-faculty.stanford.edu/~knuth/lp.html>] .