Adaptive Workflows with Syrup

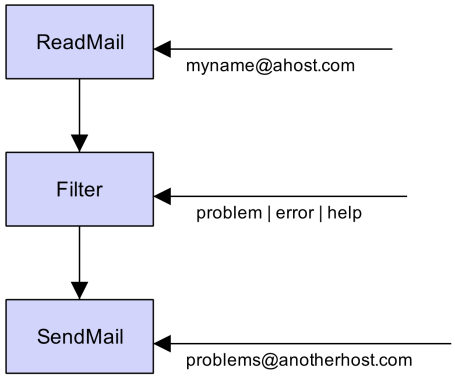
Syrup is an adaptive Workflow system with a difference. Like any other Workflow solution, Syrup can be used to describe the tasks, procedural steps, required input and output information and tools needed for each step in a business process [1]. To be able to do this, Syrup provides five basic concepts: Tasks, Links, Workflows, Workers and the WorkSpace. Additionally, the minimal core is build in such a way that integrating Syrup into an existing infrastructure poses a minimal challenge to developers.

Syrup can overcome the von Neumann bottleneck that stops traditional software systems from scaling [2]. It does this by strictly separating the specification, identification and execution phase of Workflows in a distributed setup. Although the phases are explicitly separated, this doesn't mean that each phase can't be expressed in terms of one another. For example, Workflows can be dynamically compiled by other Workflows in Syrup. This Lisp-like flexibility gives developers a chance to maximize concurrency and the distribution of Workflows even further [3].

Note that Syrup doesn't follow the more complex standards such as Wf-XML, BPML and XPDL [4]. Instead it provides the basic building blocks to implement any of the standards - and hopefully more.

Example Workflow

The following example shows a simple mail filter to give a first taste of what Workflows are about.



All mails of 'myname' are read by ReadMail. These mails are pushed through and filtered on the keywords 'problem, error and help'.
The mails that contain these keywords are pushed through SendMail and send to 'problems@anotherhost.com'.

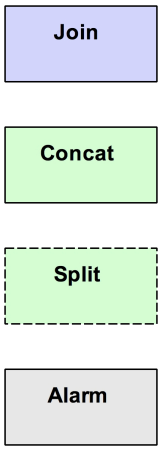
While this example may be considered not very realistic, it is already very close to a real implementation. Indeed Workflow solutions can help bridge the gap between specification and implementation, not only at the initial phase but also throughout the whole development cycle. Ideally, Syrup could serve as an intermediate (visual) language that can be talked by both developers and the business [5].

To see how to build Workflows with Syrup, the basic building blocks are introduced first.

Task

A Task is a static description of how work must be done. It can describe a calculation, transformation or any other programmable function. Tasks are managed by Syrup and executed by Workers.
One or two inputs are given as parameters before a Task can be executed. These input values are determined at runtime and they are typically produced by other Tasks.
After execution, the Worker may have produced one or two outputs based on the Task's static

description and inputs. At the same time, the Task's inputs may have been consumed so that the Task will be ready for a next round of execution.



Tasks are graphically depicted as filled squares with a certain color and label. The label describes the function and the color indicates the state of a Task:

The green color indicates the state `EXECUTABLE`.
The blue color indicates the state `WAITING`.
The grey color indicates the state `DONE`.

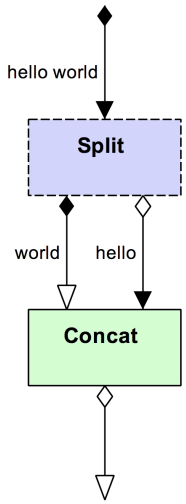
A Task also has a fixed type, indicated by its boundary line that is either dotted or solid. Type `OR` is indicated by the dotted boundary line. Type `AND` is indicated by the solid boundary line.

Tasks can turn from `WAITING` into `EXECUTABLE` and vice versa during the execution of a Workflow. In contrast, when Tasks are `DONE` they will stay `DONE` forever. A Task is `WAITING` when its inputs are not available or when it is prohibited to generate outputs. Finally, when a Task is in state `EXECUTABLE` it may consume inputs and generate outputs when executed.

Tasks of type `OR` are ready for execution if *either* of the two inputs is available. Type `AND` Tasks can only execute if *both* inputs are available. The availability of inputs and the prohibition for Tasks to generate outputs is fully determined by Links.

Link

A Link is a connection between the output of one Task (the source) and the input of another (the destination). Another possibility for a Link is to have only one Task at its ends. This type of Link will typically mark the initialization or finalization of a Task or even a whole Workflow. Links can be in two states: they are either `FULL` with information or they are `EMPTY`. It is only when the destination Tasks of Links are executed, that Links can turn from `FULL` to `EMPTY`.



Links are graphically depicted as lines connecting squares. The arrowhead shows the direction of the Link - *from* one Task *to* another Task.

The 'from' side of a Link can be either the first output (white diamond) or the second output (black diamond). The 'to' side of a Link can be either the first input (white arrow) or the second input (black arrow).

The label indicates that the Link is `FULL` with data. This can be either the data itself or a reference to the data (i.e. an URL).

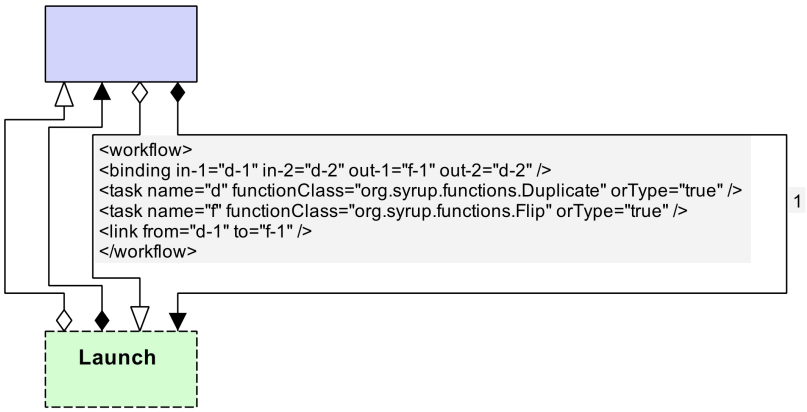
Consequently, a Task can only turn into `EXECUTABLE` when *all* its outgoing Links are `EMPTY`. After execution, a Task may have consumed input - clearing its (full) ingoing Links - *and* it may

have produced outputs - filling its (empty) outgoing Links. Following this pattern for each executable Task, it becomes clear that Links manage the data flow between Tasks [6].

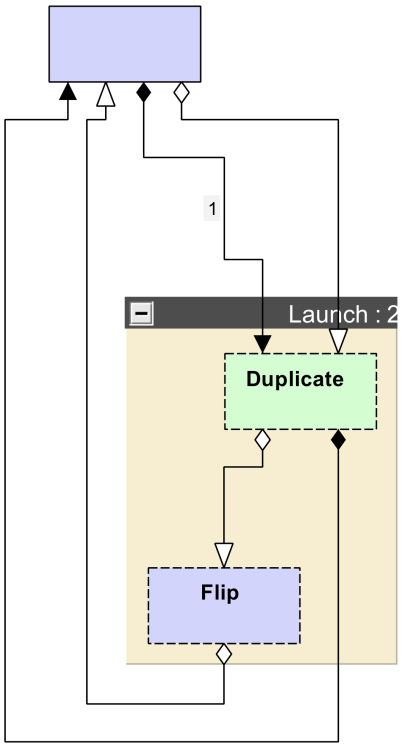
Workflow

Workers execute Tasks to consume inputs and produce outputs, but they can also do something more special: create new Workflows. A Workflow is essentially a network of Tasks and Links between them. New Workflows can dynamically be added during the run of a Workflow or adapted to meet new business requirements. Adaptive Workflows can be regarded as the most powerful feature of Syrup.

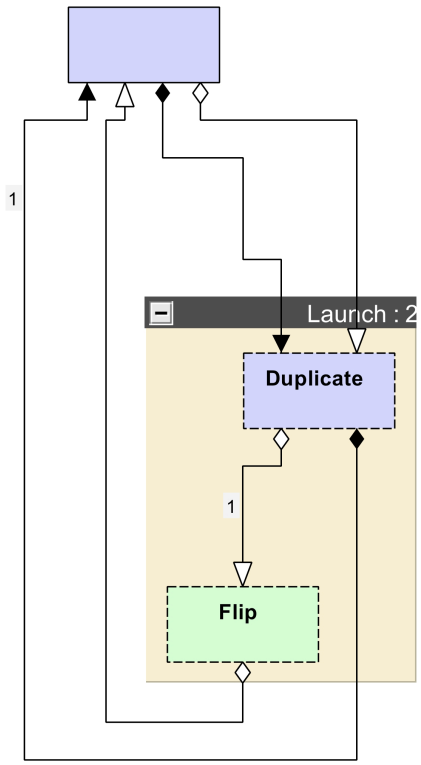
The following example shows this feature in more detail:



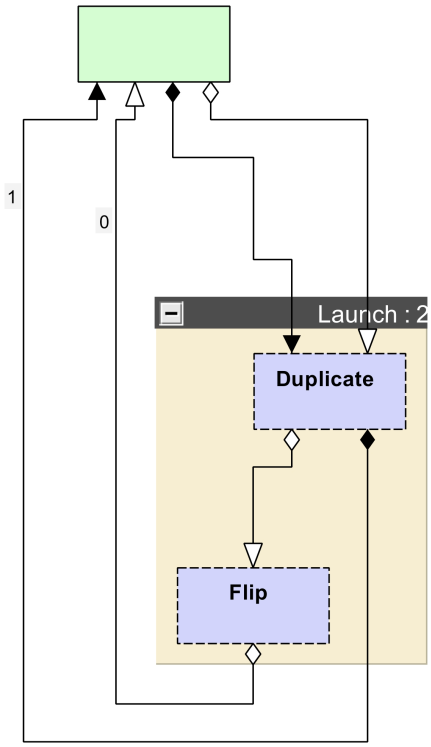
1: The Launch Task is ready to take a XML Workflow description and expand the description into a Syrup Workflow.
The XML description declares the Tasks ('d' and 'f') and the bindings and links. The references to Tasks (d-1, d-2, f-1 and f-2) contain suffixes -1 and -2 to specify which input or output is taken (first or second).



2: The Launch Task has been executed and expanded to a new Workflow. The Launch Task's old inputs and outputs are now bounded to the new Duplicate and Flip Task. The surrounding box 'Launch' indicates that the Launch Task became the parent of the new Tasks.



3: The Duplicate Task has duplicated the '1' input to both its outputs.



4: The Flip Task has flipped the '1' input to output '0'.

Worker

The Worker is the entity that does the actual work described by a Task. Workers are needed because Syrup is only concerned with the administration and coordination of Tasks and not with their execution. Workers use Syrup to know what Tasks need to be executed and in what order. Workers also deal with the efficient use of computational resources and the scheduling of Tasks.

Besides interacting with Syrup, Workers may decide to interact with other systems while executing Tasks (databases for instance). They can even decide to collaborate with other Workers, using only the 'directory' service of Syrup. By using this single service, efficient Peer to Peer (P2P) solutions could be build on top of Syrup without much effort.

WorkSpace

This is the top-level container holding the Tasks managed by Syrup - the ultimate parent of all Tasks. It is the WorkSpace that enables Workers to query Tasks, mark them for execution or commit results. The interaction between Workers and the WorkSpace is what makes a well behaved system. Because the Workspace holds and manages all the Tasks, most of the time it can be regarded synonymous to 'Syrup' itself.

Determinism

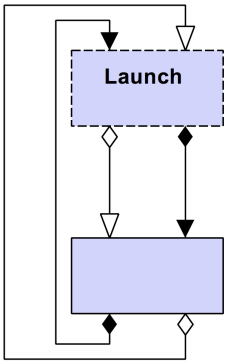
Syrup enforces that Tasks executed by Workers behave in a deterministic way - output should solely depend on the given input (pure functions) [7]. This property is essential for Syrup to ensure that faulty executions, due to crashed Workers, can be rolled back safely.

If this was not so, Syrup would be held responsible for actions based on the internal state of Workers. Because Syrup cannot possibly know everything about the internal state of Workers, this functional restriction can be justified.

Non-Determinism

If Syrup were purely deterministic, how then can it react to external events? External, real world events are notorious for their non-determinism - they just happen.

Syrup provides two special inputs to receive external events and two special outputs to generate external events. In fact, it is only via these special inputs and outputs that the Syrup system can be bootstrapped, for instance, to receive Workflows from the external world.



The two ingoing Links of the WorkSpace can be filled to signal events from the external world (represented by the non-labeled box). The two outgoing Links can be consumed to publish Syrup events to the external world. Only specialized Workers are allowed to do this.

There is another, more subtle kind of non-determinism. This is introduced by type `OR` Tasks that may depend on the *relative* timing between two inputs (comparable to race-conditions) [8].
For example, consider a Task that has the following behavior: if the earliest available input is the second, the output will be 1. Likewise, if the earliest input is the first, the output will be 0.

Design and Implementation

Environment

The default implementation is done in Java, but is not restricted to Java alone. The implementation is also able to control Perl, shell scripts or C++ executables in a distributed environment. This makes Syrup ideally suited for developing system integrations [9].

Distribution

Remember that Syrup itself does not execute Tasks. It only manages the flow between Tasks and the creation of new Workflows. However, Syrup does track all the Workers that have requested the execution of Tasks so that Workers will cooperate nicely.

Syrup can track thousands of distributed Workers that execute Tasks concurrently, making distributed computation a reality. It is also designed in such a way that crashed Workers, network failures or other exceptions do not influence the outcome of a Workflow [10].

Persistence

Tasks are managed with the aid of a persistent backing store so that no Task or execution result is ever lost during the lifetime of a Workflow. Persistence is desirable when Workflows take weeks or even months to complete or when (partial) failure is not an option.

The default implementation uses (My)SQL [11] to store and retrieve Tasks. But other, potentially more distributed stores such as JavaSpaces [12] could be used to achieve even higher performance.

Scheduling

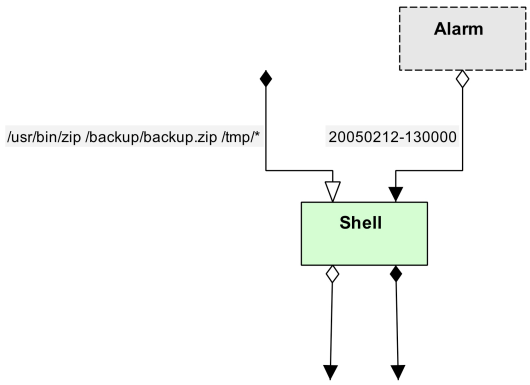
In the UNIX environment, the cron utility is primarily used to schedule jobs on a single host. Note however, that cron doesn't prevent the loss of scheduled jobs when the machine is temporary offline [13]. Alternatively Syrup could replace cron so that jobs can be scheduled reliably on a single host but also across multiple hosts.

Example crontab

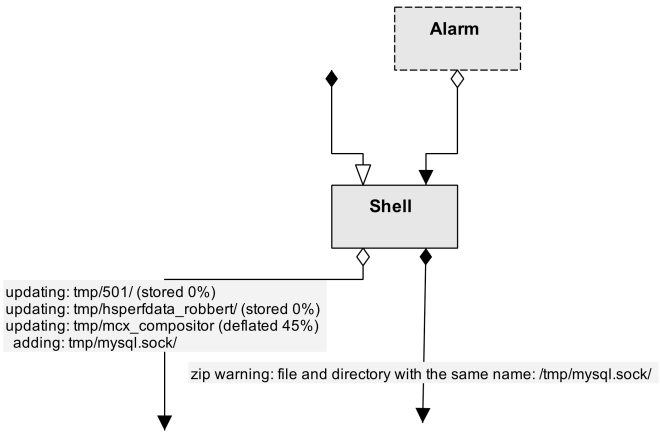
Consider the replacement of cron entries by Syrup. The following approach could be taken: For every cron entry, create a Workflow that includes the following two Tasks:

- a Task that can track time (for every workday ....) and signals when a certain deadline has passed.
- a Task that runs the shell command that would have normally be run by cron. After it receives the time signal from the first, it executes the shell command.

The following diagrams show the replacement of the following cron entry: '00 13 * * 1-5 /usr/bin/zip /backup/backup.zip /tmp/*'.



The Alarm has signaled that the deadline has passed.



The Shell Task has run '/usr/bin/zip /backup/backup.zip /tmp/*'. After execution, it produced stdout on the first output and stderr on the second output.

For Workflows to be executed by a Worker, a cron entry (can be on different hosts) has to be added to start the Worker and to run it frequently and reliably throughout the day. Each time a Worker is launched by the cron scheduler, it will fetch Tasks from the Workspace and execute them. By adding more and more Workers to the crontab, higher levels of parallelism can be achieved without much effort.

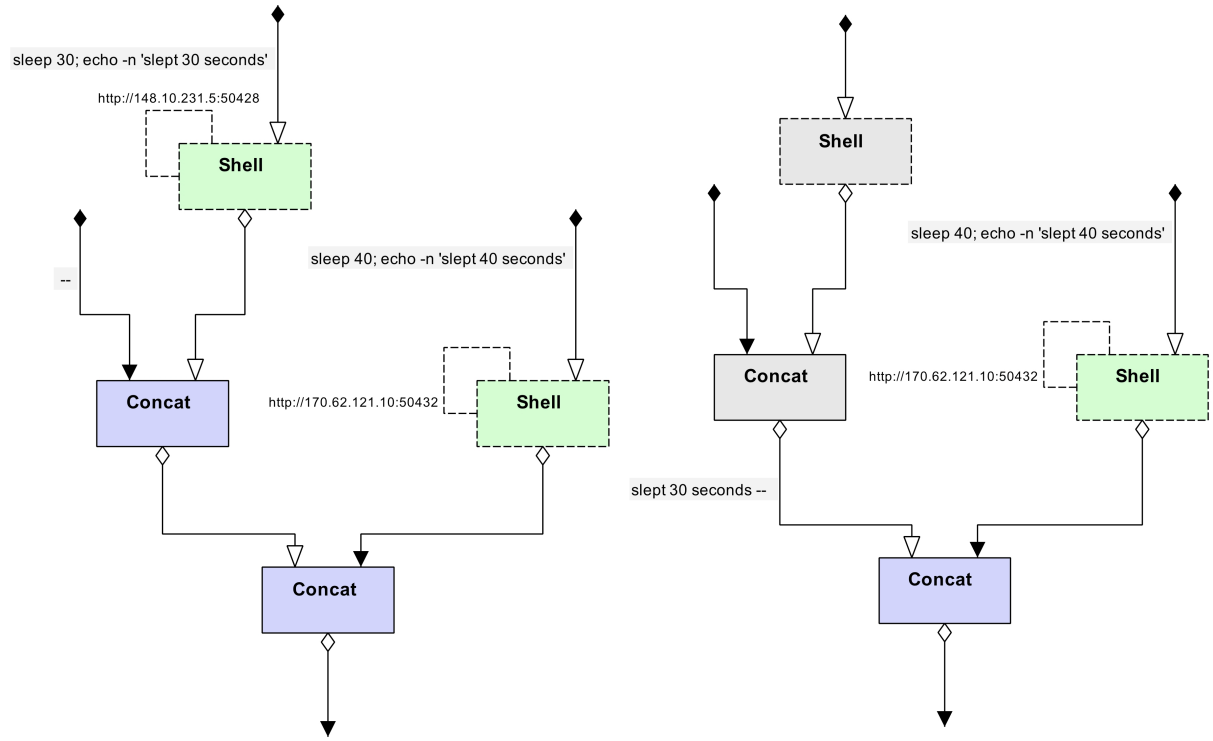
Syrup processing

Tasks can be executed by Workers, but who runs the Workers? As the previous cron example showed, Workers are typically run by a concrete operating system but could - in principle - be humans as well. Workers are build to run forever (at least during the lifetime of their hosts) possibly having executed thousands of Tasks that were taken one-by-one from a WorkSpace.

Although Workers are considered to be the active Agents that have the ability to process something, they are not only concerned with executing Tasks. In fact, Workers have to first find a WorkSpace to be able to do something useful. And after the WorkSpace has been found, a Worker has to query it for Tasks to be executed. So, besides execution, Workers have to do some additional processing in order to find and retrieve Tasks from a WorkSpace.

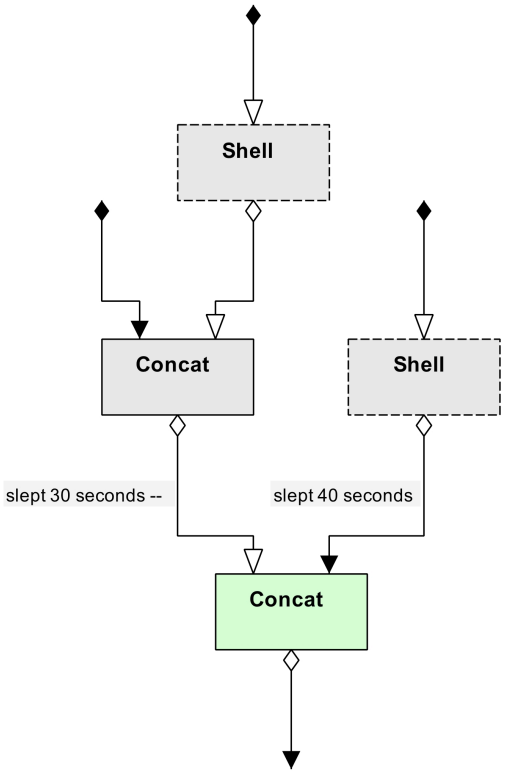
Query and select

Idle Workers can start querying the WorkSpace for Tasks waiting to be executed. When such Tasks are found, they are carried out in a specific order - based on the Worker's internal scheduling policy.

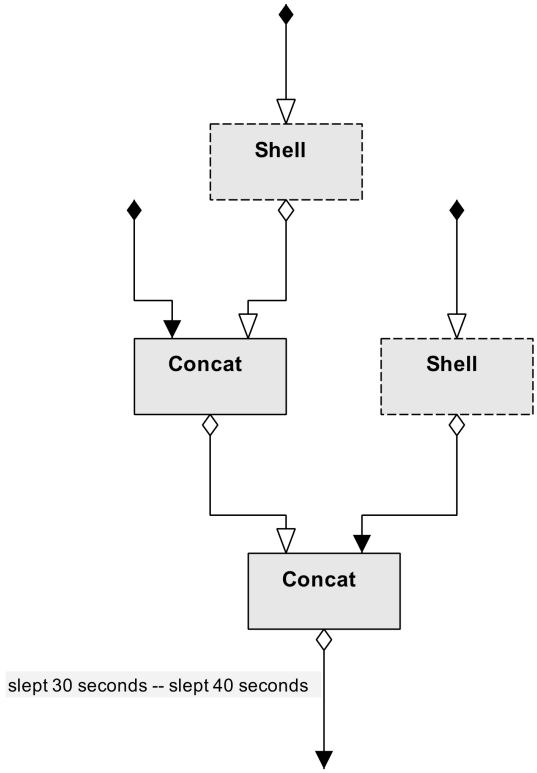


1: There are 2 Shells being executed by 2 different Workers concurrently. The label of the dotted line indicates the (HTTP) address of each Worker. The first Shell will sleep for 30 seconds and then output 'slept 30 seconds' to stdout. The second Shell will sleep for 40 seconds and then output 'slept 40 seconds'.

2: The first Shell is finished while the second is still in progress. The Concatenation of the strings 'slept 30 seconds' and ' -- ' has finished. Note that Workers that execute small Tasks (~1 second) such as Concat will publicize their address for the same short period of time (~1 second). It is only during the execution of medium to big Tasks that addresses will be visible to other Workers.



3: The second Shell is finished and produced 'slept 40 seconds' on stdout.



4: The string 'slept 30 seconds -- ' is concatenated with the string 'slept 40 seconds'.

Before execution of any single Task, a Worker must notify the WorkSpace that it wants to do so. This notification has two effects:

First, the Worker identifies itself as a 'reachable' resource by creating a unique (communication) address where the Worker can be reached. During execution, the Worker is expected to respond to any request send to that address. Second, the communication address is stored in the WorkSpace. So, for each Task that is in progress, there will be an associated and known address.

Given an address, other Workers are able to check whether the executing Worker is still active. If the Worker is not responding to requests, other Workers may decide to take a turn in executing the Task.
The preferred address type that is used among Workers is an URL referring to an HTTP address. The choice of using the HTTP protocol is convenient because HTTP requests are synchronous and are thus more responsive when compared to the email protocol for example.

As the previous examples have shown, Tasks can be in one additional state: `EXECUTING`. This state is indicated by the green fill color of the `EXECUTING` Task and a self-referencing dotted edge. The corresponding label states the URL address of the Worker executing the Task.

Execution and failure

As previously mentioned, Syrup is resilient towards network failures or power outages. Still there is one type of failure that is difficult to catch, in particular the case in which Workers

may have crashed or are otherwise blocked. This problem is closely related to identifying (from the outside) which processes are still doing something useful or which of them are in some kind of infinite loop.

For example, a Worker preempted (put to the background) by the operating system is essentially doing nothing but hasn't exactly failed either. The Worker just has to wait for another time-slice to continue. The difficulty is that the (preempted) state of one Worker is hard to assess by others. In fact, the only way to check the state of a Worker is by sending it a request. When no reply is received within a certain amount of time, the associated Worker is considered to be crashed and will be removed from the Workspace as an active Agent.

This scheme has one drawback. A Worker could possibly still have been operational but was somehow not able to respond to requests (for example, because of network failure). And when such a 'disconnected' Worker tries to commit the execution result to the Workspace there will be an error because some other Worker has taken control over the same Task (while it was disconnected). In this rare case a Task may have been executed twice (or more) by different Workers after subsequent failures (although only one result is committed to the WorkSpace). In general, distributed systems cannot prevent redundant executions without incurring great communication costs. Syrup also doesn't prevent redundant executions. Instead it adopts another algorithm that tries to keep redundant executions to an absolute minimum [14].

When Tasks are to be executed multiple times however, it is important to know that they behave like pure functions (see Determinism). The alternative - impure functions - should be avoided because they typically modify state outside the realm of Syrup (like modifying database records or transferring money). Such stateful programs usually require that their modifications be applied once (and only once) to ensure overall correctness.

In contrast, the proper and exclusive use of pure functions will always yield the same outcome regardless of how many times they are executed.

Temporal data

If state cannot be avoided it is best to use temporal databases. Temporal doesn't mean they are thrown away after use. Instead they have the unique property that state cannot be *changed* in time but only *added* (with explicit timestamps). Practice has shown that if Syrup and temporal databases are married, they are able to produce fully accountable, easily distributable and very scalable systems [15].

Controlling complexity

It is the dream of the author that, when dealing with increasingly more complex systems, solutions like Syrup and temporal databases can give programmers and business people more insight and less to worry about. And if not, Syrup could at least make the building and running of software more fun.

References

[1] T. Bayens: The state of Workflow.
[2] J. Backus: Can programming Be Liberated from the von Neumann Style?
[3] P. Graham: On Lisp.
[4] W. van der Aalst: WorkFlow patterns.
[5] R.E Horn: Visual Language and Converging Technologies in the Next 10-15 Years (and Beyond).
[6] J.P. Morrison: Flow-Based Programming.
[7] J. Hughes: Why Functional Programming Matters.
[8] E.A. Lee, T.M. Parks: Dataflow Process Networks.
[9] Sun microsystems: The Java Language Specification.
[10] F.C. Gartner: Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environment.
[11] MySQL: http://www.mysql.com
[12] Sun microsystems: JavaSpaces (TM) Service Specification
[13] UNIX manual pages: crontab - user crontab file.
[14] G. Malewicz: Distributed Scheduling For Disconnected Cooperation.
[15] F. Bajers: The Consensus Glossary of Temporal Database Concepts.