Peer-to-Peer Algorithms for a Distributed Chemical Machine

Raphael Proust

Peer-to-Peer Algorithms for a Distributed Chemical Machine

Raphaël Proust[*], Cédric Tedeschi[†]

2011

[*] ÉNS Cachan, Antenne de Bretagne
[†] Irisa, Université Rennes 1/Inria, équipe MYRIADS

Summary

The present document reports the work done during the first four months of an internship hosted by the MYRIADS team under supervision by Cédric Tedeschi.

The aim of the internship is to study various algorithms that can help to create a distributed runtime for the Chemical Programing paradigm. The Chemical Programing model is an intrinsically parallel, non deterministic programming paradigm offering abstraction and expressiveness. It is based on a metaphor between rewriting rules and chemical reactions: bits of data are rewritten to other bits just as molecules are replaced by other molecules. The process by which a reaction happens in this unusual paradigm is three fold:

- pattern matching is used to select the appropriate number and type of molecules for a given rule;
- the rule's condition is tested on the set of selected molecules; and, if the test is successful,
- selected molecules are substituted with new ones.

During the internship, a model of Distributed Chemical Machine has been developed. This model includes a modified reaction process and a distribution scheme. A simulator has been implemented so as to test some features of the Distributed Chemical Machine.

The report gives context to the subject, explaining why a distributed runtime for the Chemical Programing paradigm is highly relevant in the Internet of Services we experience daily. Chemical Programing is then described from both an intuitive and formal point of view. A section thoroughly details the specification of our Distributed Chemical Machine. Finally, the simulator for this model is presented along with results it helped obtain. The conclusion gives insight in the future developments we expect about distribution of the Chemical Programing paradigm.

Contents

1 Introduction

1.1 Web Services and SOA

Since its creation, the Internet has been evolving, both in its hardware structure [11] and its usage [18]. From a collection of static documents loosely linked together, the World Wide Web (or simply the Web) has changed greatly. Several key steps have already been identified and given names. Each of them offering new possibilities to users. At first was a set of static HTML documents directly copied from disk to the client's connection. Then appeared dynamically generated content with template systems and databases. This was further enhanced by the use of JavaScript which allowed code to be run on the client side. Recent developments are geared toward an applicative Web [17]. That is, a Web where pages are full featured user interfaces for underlying applications. Famous examples include Webmails such as Gmail[1], collaboration suites like Zimbra[2], micro-blogging platforms a la Twitter[3] and task management applications such as Remember The Milk[4].

The user experience drives changes in backend organization and so Web applications rarely run on a single machine. A great many[5] of them uses a service oriented architecture (SOA) model [15]. It means that Web Services interact with each other through the network to bring functionalities to the user. Web Services are said to be *composed*. A classic example deals with price comparison: a Web Service gathers data from several commercial sites to present a wide choice to the user. Data from commercial sites is obtained through the aggregation of multiple Services, and an external check-out Service can later be used to process payment. The overall result is a *composite* Web Service. In the Internet-of-Services model, composition has a central role: it is how programming is done in the framework and how Web applications are built.

Nowadays, composition operations are generally specified using XML description files[12, 13]. The obtained workflow is often executed via a central entity orchestrating the different services used. Alas centralization leads to poor scalability and the presence of a single-point-of-failure pattern. Moreover, XML file specifications are quite rigid and make dynamic composition difficult. Both the file specification issue and the workflow execution problems are observed in platforms such as Pegasus-DAX[6] and Taverna-Scufl[7].

We believe a better solution exists, one based on a programming model which verifies a few needed properties: scalability and dynamicity, of course, but also self-management, expressiveness and evolvability.

[1] `http://mail.google.com`
[2] `http://zimbra.com`
[3] `http://twitter.com`
[4] `http://rememberthemilk.com/`
[5] Map mashups, online shops with external checkouts, travel agencies, twitterfeed, etc.
[6] `https://pegasus.isi.edu`
[7] `http://www.taverna.org.uk`

1.2 A New Paradigm

As explained, evolution of the Web makes application development difficult. Indeed developing a Web application requires the use of multiple tools. Existing tools needed for the specification and execution of Web Service composition are not suited. We think the use of the Chemical Programing paradigm for both the specification and the execution of Web Service composition[8] would both lighten the developer's burden and improve scalability and fault tolerance as explained in [7].

In order to benefit fully from this paradigm shift, it is necessary to be able to run Chemical Programs in a distributed fashion. The present work is focused on this issue.

[8]While it has not been the focus of the internship, we give a possible expression of Web Service composition in the Chemical Programing paradigm in Annexe A.2.

2 Chemical Programming

Chemical Programing is a paradigm based on a natural metaphor. Nature inspired techniques have been used in different areas of Computer Science, most notable examples include ant colony optimization, genetics algorithms and well known neural networks. We here describe Chemical Programing, both as a metaphor and a formal model, and list a few interesting characteristics.

2.1 Chemical Programing in a Nutshell

2.1.1 Informal Description

The Chemical Programing paradigm is based on a metaphor associating data with molecules floating in a solution and programs with reactions. The runtime for Chemical Programing is comparable to the Brownian Motion or, better yet, a magnetic stirrer. Hence, a program is simply a description of a transformation process applied on data. The input of a program is the initial solution.

A reaction rule is totally determined by the set of reactants, a reaction condition and a set of products. The usual notation for a reaction in chemistry is $r_1 + \ldots + r_n \xrightarrow{cond} p_1 + \ldots + p_m$ where the r_is are the reactants, *cond* is a condition on the reactants and the p_is are the products[9]. We will use a different style when describing a computer program, e.g. `max: replace (x,y) by x if x >= y` with `x` and `y` being the reactants of a reaction conditioned[10] by `x >= y` and producing `x`. This program works as follows: a pair of numbers is replaced by its greater component. For the program to complete, the Chemical Machine — i.e. the runtime, our metaphorical magnetic stirrer — has to apply the rule as long as it is possible to do so. When no rule can be further applied, the solution is said to have reached *inertia*.

As the metaphor would suggest, execution of a Chemical Program is not deterministic. The order in which the reactions are carried out cannot be known by the programmer. While some implementations may introduce determinism, the semantic of the language is *not* deterministic. Another feature inspired by the metaphor is the concurrency: two — or more — reactions can happen concurrently, provided they have disjoint sets of reactants. In the aforementioned `max` program, only the interaction between the integers is specified. The order the comparisons and eliminations are performed in is left for runtime to decide. The runtime can eliminate several molecules concurrently.

The Chemical Programing paradigm is free from some of the limitations the metaphor could lead to. Indeed, contrary to real chemical processes, there is no need for the reaction rules to be balanced. The ability not to balance rules makes it possible to filter out elements as demonstrated in the aforementioned *max* program. Strictly speaking, it is not even necessary for a reaction to

[9] For practical reasons, chemists use stoichiometric coefficients to shorten notations. We will not go into this much details as the Chemical metaphor is just a metaphor.

[10] When the reaction condition is tautological, it can be omitted altogether.

produce anything — as showed in `odd: replace (x,x) by _`, where only elements with an odd number of occurrences in the solution are kept. Similarly, a reaction may produce more molecules than it consumes.

2.1.2 Formal Description

While the informal description makes it possible to get an idea of what Chemical Programing is, we here take a more theoretical approach in our explanation. Several formalizations of the Chemical Programing paradigm exist. We give a short description and point the reader to complete references for a few of them.

All the formalizations and implementations that we know of use the *multiset* as the supporting data structure for chemical solutions. A multiset is similar to a set, except that elements can appear more than once — the number of occurrences of an element is called its *multiplicity*. Hence a set is but a particular case of multiset were elements occur either once or not at all — elements can have a multiplicity of either 1 or 0. Standard operations on sets — union, intersection and the like — can be modified so as to apply on multisets. Annexe A.3 goes further into the multiset formalism and lists a few of the adapted operations. Different extensions — such as negative or infinite multiplicities — can be devised and given meaning to. Several of them are formally described in [14].

Γ: The Γ model is thoroughly described in Banâtre et al. seminal paper [3]. It is a simple formalism for the Chemical Paradigm. Simplicity refers to the limited number of language constructs: over a multiset, one can use a pair (R, A) where R is a *reaction condition* and A is an *action*. Reaction conditions are predicates over tuples and actions are partial functions from tuples to multisets. The model is defined by Γ, its only operator.

$$
\begin{aligned}
&\Gamma(R, A)M = \\
&\qquad \text{if } \exists\, \{m_1, \ldots, m_n\} \subset M \text{ such that } R(m_1, \ldots, m_n) \\
&\qquad \text{then } \Gamma(R, A)\,(M - \{m_1, \ldots, m_n\} + A\,(m_1, \ldots, m_n)) \\
&\qquad \text{else } M
\end{aligned}
$$

Intuitively, when applying a reaction rule — defined by its condition R and its action A — on a solution M, either it is possible to apply the rule or inertia has been reached. The former case evaluates to the recursive application of the same reaction rule to the modified solution while the latter evaluates to the solution itself.

Calls to the Γ function can be chained so as to apply successive transformations. It is also possible to generalize Γ to accept several condition-action pairs. We do not give here such a generalization.

CHAM: Introduced by Berry and Boudol in [4], the CHemical Abstract Machine (CHAM) is inspired by both chemistry and biology. The primary addition with respect to Γ is the notion of *membrane* isolating a sub-solution. Sub-solutions can communicate through a system of *airlocks* which makes it possible to express the crossing of a membrane by a molecule. The increased complexity is justified by the increased expressive power, most notably the possibility to translate CHAM programs to and from process calculi expressions [4].

The CHAM's concepts are similar to those of P-systems introduced by Gheorghe Păun in [2]. In P-systems, rewriting rules and elements on which they apply are organized into a hierarchical set of membranes or cells evolving concurrently. With a lot of inspiration from biology — especially cell biology — P-systems, like the CHAM, allow for richer data structure than the flat solution of Γ.

HOCL: The Higher Order Chemical Language (HOCL) is a full featured computing language built atop an extension of the Γ model called γ-calculus. γ-calculus introduces several concepts, most notably higher order — making Chemical rules first class elements similar to molecules — thus giving programmers the ability to implement rules tampering with other rules. Another feature of the γ-calculus is the sub-solutions. Sub-solutions are first class components isolated from the rest of the solution. A sub-solution can be opened whenever it reaches internal inertia. Sub-solutions are somehow similar to CHAM's membranes with notable differences in the way they interact with one another.

The MYRIADS team is working on a HOCL to Java translator. A partial description of the internals of its code can be found at the end of Yann Radenac's thesis [14]. Because the translator targets the Java language, it is possible to use Java objects directly as molecules. This is achieved by creating a class for the desired molecule and calling its constructor in the initial solution.

Complete reference for both HOCL and γ-calculus is available in [14].

2.2 Chemical Programing Benefits and Hindrances

Among other things, the Chemical Programing model brings high level of abstraction and expressiveness to the programmer. High level, expressive languages tend to reduce code size, thus decreasing the maintenance and debugging burden. These two features are essential to anyone who believe that programmer time is more precious than hardware time[11].

However, this abstraction has a cost. Combinatorial explosion arise when a Chemical Machine tries to find a match among a large number of molecules. Moreover, rules arity is arbitrary. Combinatorics considerations allow us to precisely quantify the number of match candidate for an n-ary rule applied on a multiset of m molecules to $\binom{m}{n}$.

[11] "Programmer time is expensive; conserve it in preference to machine time" - Eric Steven Raymond in The Art of Unix Programming

This complexity issue can be tackled in several different ways. First of, it is possible to filter out several match candidates in a single test. This method, detailed in [9], consist in having a reaction condition $\Phi(m_1, \ldots, m_n)$ rewritten to an equivalent conjunction $\Phi_1(m_1) \wedge \Phi_2(m_1, m_2) \wedge \ldots \wedge \Phi_n(m_1, \ldots, m_n)$ of n clauses of different arities. Failing an intermediate test Φ_k makes any attempt to complete the tuple unnecessary, thus eliminating numerous candidates.

Another possible approach to this combinatorial problem consists in parallelizing the match process by having several match candidates tested concurrently. It is necessary in this case to make sure that a single molecule does not participate in several concurrent reactions. Two distinct approaches are available in [3] and [9]. The former uses a complex — but proved correct — lock system while the latter uses message passing instead of shared memory to ensure that each molecule is only accessible by one processor.

We chose to tackle the combinatorial issue by distributing the Chemical Machine on several computing devices. This is arguably better than parallelisation on a single machine as it is possible to add and remove resources dynamically.

3 Distributed Chemical Machine

As mentioned before, a Chemical Machine is a runtime for the Chemical Programing paradigm. It is responsible for managing the solution, finding applicable reactions, extracting the necessary molecules from the solution, applying computations and introducing new molecules into the solution. After discussing the mean of distributing an application we will detail the simplifications we assume for the model. Following parts will give a simplified version of our Distributed Chemical Machine and list some improvements that allow for better performance.

3.1 Distribution

Distributing an application can take many forms. As specified in the internship subject, we try to avoid any unnecessary centralization by not considering server-client based solutions, thus focusing on P2P architectures. Mixed scenarios, where some nodes have more responsibilities than others, also exist — e.g. BitTorrent [6]. There are several ways to structure P2P systems. A P2P architecture usually provides an *overlay* — that is, a logical abstraction of the physical underlying network. On the Internet, every machine can send data to any machine it knows the IP address[12] of. As only part of this network participate in the P2P system, it is useful to build a logical network linking the nodes together. Moreover, a logical overlay abstracts out the heterogeneity of the network. Content distribution through P2P systems are surveyed in [1].

We chose to structure — or rather not to structure — the nodes of the Chemical Machine through gossip. Gossip is a minimalistic communication paradigm in which nodes exchange information in a chaotic fashion. It provides, among other things, an overlay building method in which each peer maintain a partial view of the whole logical network. This view is a list of known peers. Nodes regularly exchange information about their view. Exchanged information is used to modify the list of known peers by merging local and received information. The system is, thus, continuously evolving. A general framework for gossip overlays is described in [5].

In a gossip overlay, it is not possible to reach a specific peer because the logical network is not stable enough to allow routing. The usefulness of this overlay can thus seem limited. However, gossip has a very low maintenance cost compared to DHTs [5]. Minimalistic, simplistic designs are generally considered to be superior in the Unix philosophy[13].

The main operation provided by gossip overlays is Random Peer Sampling (RPS): the possibility to reach a peer, without the capability to purposely chose which one. Our Chemical Machine does not need routing but makes heavy use of RPS. We leverage the volatility of gossip overlays as a source of randomness.

[12] It is not possible to send data directly to a machine that has no IP address. This includes computers behind NAT.

[13] "Rule of Simplicity: Design for simplicity; add complexity only where you must." - Eric Steven Raymond in The Art of Unix Programming

Indeed, gossip overlays' RPS have been shown [10] to expose true randomness — granted a few properties on the view exchange method. This randomness in RPS can be leveraged so as to provide randomness to the Chemical Machine.

In a Chemical Programing setting it is possible to guarantee the progress of a reaction by imposing a few properties to the runtime. The default assumption is to consider that reactions are chosen by a potentially malicious adversary. However, we can also ask for the system to be:

Random reactions and reactants are chosen in a truly random way, thus providing probabilistic guarantees.

Weakly fair a possible interaction eventually happens, thus forcing the computation to actually make progress.

Fair when a reaction is possible an infinite number of time, it happens an infinite number of time, thus, not only forcing the computation to make progress, but also preventing some executions to occur.

While no proof was made on this specific issue — mainly out of time concerns — the rationale behind the gossip choice is clear: RPS provides randomness which can be leveraged. As molecules will be exchanged between nodes participating in our machine, providing randomness in exchanges can provide randomness in molecules motion. This specific use for gossip adds up with the benefits induced by simplicity such as low maintenance cost and easy development and debugging.

3.2 Hypotheses

In order to understand the built Chemical Machine it is necessary to introduce a few concepts and state beforehand the context in which the machine is supposed to run.

Metadata: The built machine handles each molecule as one Datum and one Metadatum. The Datum contains all the information the molecule carries. The Metadatum includes type information — indicating what the molecule is — and Data retrieval information — indication where the Data is and how can it be loaded. Metadata is used in the pattern matching process when the machine selects a tuple of molecules. The Data is used both for testing the molecules and performing the necessary substitution.

We explain the need for this division as follows. When working on a single computer, it is possible to use molecules as indexes and offsets, but also as file descriptors[14] which contents are used in the substitution process. On a single machine the content of a file whose descriptor is used in a reaction can be accessed transparently and the molecule can be said to represent a file while

[14] And of course any other value a computation requires, be it string, character, elements of a product or sum type, lists, arrays, etc.

actually being a file descriptor. For a Distributed Chemical Machine however, this transparency is not necessarily suitable. When working on files, it is better to have nodes transmitting each others Metadata about them rather than their potentially heavy content.

Storage Device: A choice was made not to try to solve all the problems raised by the distribution of a Chemical Machine but rather to focus on the definition of a base model. For this reason, we consider granted the existence of a storage device guaranteeing a number of interesting properties. It is necessary the storage device provides the following functionalities and a remote access to them:

- `fetch`: load data from the storage. Received molecules are locked for the use of the requesting node only. Further attempts will be met with `Unavailable` until molecules are `released`.
- `release`: makes locked molecules available for other nodes.
- `replace`: deletes a set of molecules and inserts another set. Note that the operation may fail atomically, meaning that either all the old molecules are replaced by all the new ones or nothing happens at all. An intermediate failure is impossible. This operation fails if the Data is not locked by the calling node.

Each of these operations fails if any one of the molecules it uses does not exist. This happens when a node makes use a stall molecule that has been `replaced` by another node.

For the Chemical Machine to benefit from distribution, the storage device should also be distributed. Distributing the storage device is a research topic by itself out of the scope of this work.

Node Failure: In our model, nodes never crash, nor do they behave inappropriately. Because of this assumption, we do not need the leave and join operations. It is an unrealistic hypothesis, especially in distributed systems were each node can be brought down or taken over unexpectedly and independently. However, because the study we carry here is unprecedented, we choose to simplify the model.

3.3 Simplified Machine

We present here a simplified version of the Chemical Machine. Improvements and details will be given subsequently.

The Chemical Machine is a network of computing devices — referred to as *nodes* or *peers* — which collaborates in order to run a Chemical Program (i.e., a set of rules applying on a multiset of data). All those nodes have access to the storage device discussed in Section 3.2. Nodes communicate with one another via direct messages in a gossip overlay.

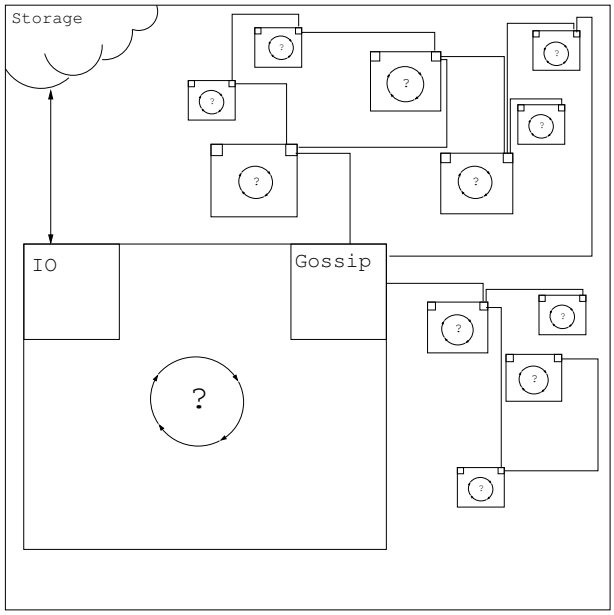

Figure 1: Global architecture of the simplified Chemical Machine.

Internally, a node sifts through a multiset of metadata, trying to find matches and processing them. The match processing happens in several distinct phases: the Data associated to the match is `fetch`ed from the storage — leading to the registration of a lock —, then the data is tested according to the reaction condition and, depending on the result of this test, either the data is `released` or `replaced` by the product of the reaction.

A node has several key modules responsible for different aspects of the computation. An I/O module is responsible for storage communications. A gossip module regularly sends Metadata to peers, thus providing the necessary shuffling. The core of the node uses the locally available Metadata in order to make the whole computation progress. Figure 1 represents the described Chemical Machine with one node being zoomed in. The core operation of a node, its internal activity, is described in Section 3.4.1.

3.4 Formalization and Optimizations

3.4.1 Diagram Representation

We can summarize the internal activity of each node of the Chemical Machine as follows:

- it finds matches in a Metadata multiset;
- it uses the storage to retrieve and lock Data;

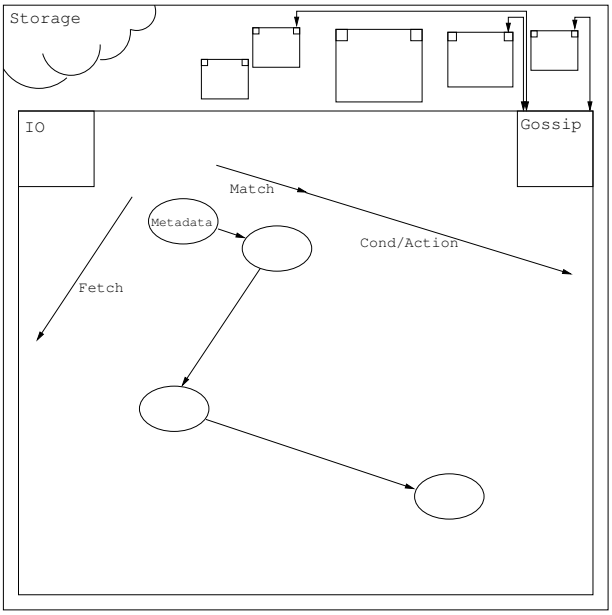

Figure 2: Coarse grain model with CPU and I/O on distinct axes.

- it performs the condition test and the substitution computation; and
- it signals the storage device so as to make the substitution global.

In this coarse grain model, we can observe that CPU and I/O operations are clearly separated. Figure 2, exposes this model using distinct axes for the orthogonal CPU and I/O steps.

Atomic steps: First of, it is possible to split some steps into smaller ones. Indeed the Data retrieval phase serves both loading and synchronizing purposes: Data is locked *and* loaded. This induces a small change in the storage interface as the `fetch` function needs to be replaced by `load` and `lock` primitives.

Orthogonally the test and computation are gathered in a single operation while it clearly perform two distinct actions. Going further in this atomizing spree it is possible to use the Metadata-Data separation to start tests early. Indeed the Metadata might carry more information than basic type and retrieval. It is not always necessary to completely trim down the information in the Data to Metadata conversion. In this case, the reaction condition can be split in two: a predicate on Metadata and another on Data.

Cache: As loading and locking are split into distinct phases, the loading of molecules can use a cache so as to reduce network usage. This is possible

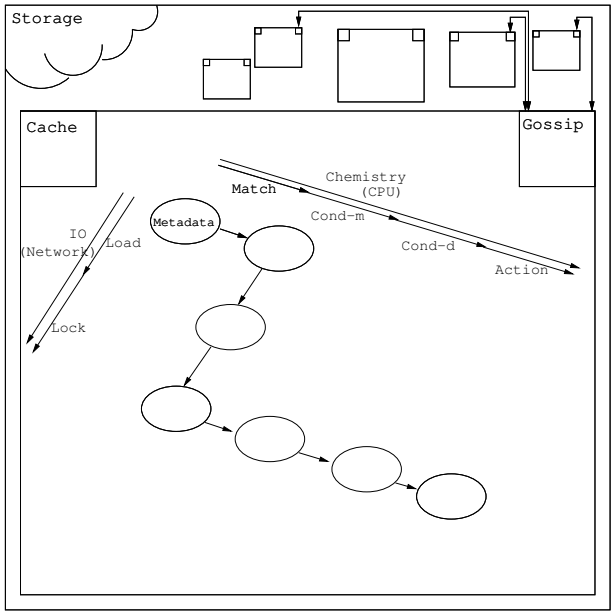

Figure 3: Atomized model.

because of the immutable — hence state-less — nature of molecules. Because it depends on the state of the whole machine, the lock operation cannot benefit from caching. The separation of loading and locking steps is what makes the cache useful.

Representation: The blunt and refined models are presented, respectively in Figures 2 and 3.

3.4.2 Strategies

CPU and I/O operations are orthogonal. Hence, it is possible to interleave the CPU and I/O operations in different fashions. Not every interleaving is permitted as some tasks depend on others — e.g. testing the Data requires it to be loaded. The possible interleavings[15] are presented in Figure 4 as a graph whose edges are the possible steps. In the context of our Chemical Machine we will call *strategy* an interleaving of the elementary steps of the refined model. We only consider valid strategies; that is, strategies respecting the dependencies of the elementary tasks.

As CPU and I/O tasks are orthogonal — modulo dependencies — some degree of parallelisation is also possible. Parallelizing the fetch and metadata test

[15]While it is possible to perform a lock on unmatched Metadata, it induces unnecessary synchrony in the Chemical Machine.

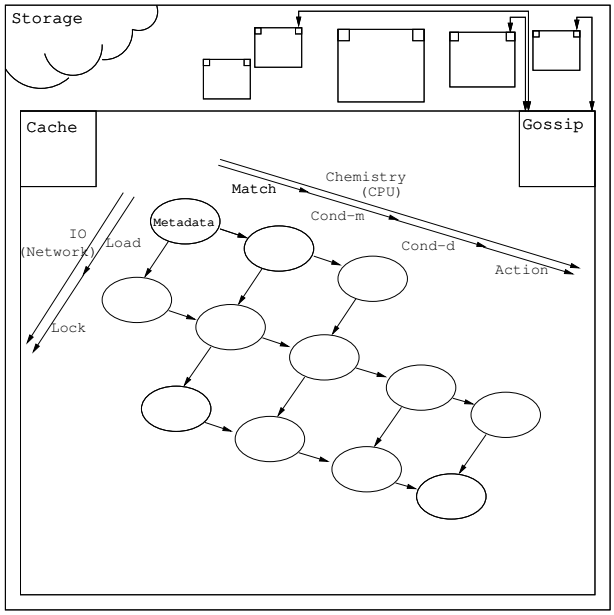

Figure 4: Possible operation interleavings.

operations consist in starting the two concurrently and waiting for both result. When doing so, an operation can be early aborted if the other's result is negative. The ability to parallelize computational and networking tasks increases the number of possible strategies. In our experiment we used three different strategies built with different rationales in mind. All three strategies are listed thereafter.

***EarlyIO*:** In this strategy, I/O operations are performed as soon as possible. We intend it for applications in which the Metadata and Data tests have very high success rate — possibly tautological — and synchronization conflicts between nodes arise often. In this particular settings it is unwise to perform costly computations that will possibly be discarded because of synchronization issues. The aim is to have the whole process fail early.

We graphically represent *EarlyIO* in Figure 5.

***EarlyCPU*:** Dually to EarlyIO, this strategy performs CPU operations as soon as possible. The settings it is intended for is dual to EarlyIO's: low success rate on Metadata and Data tests and sparse conflicts. Heavy weight Data increases even further the cost of I/O operations and thus the associated potential wastage. In order to fail as early as possible, performing CPU tasks first is preferable.

This strategy is presented in Figure 6.

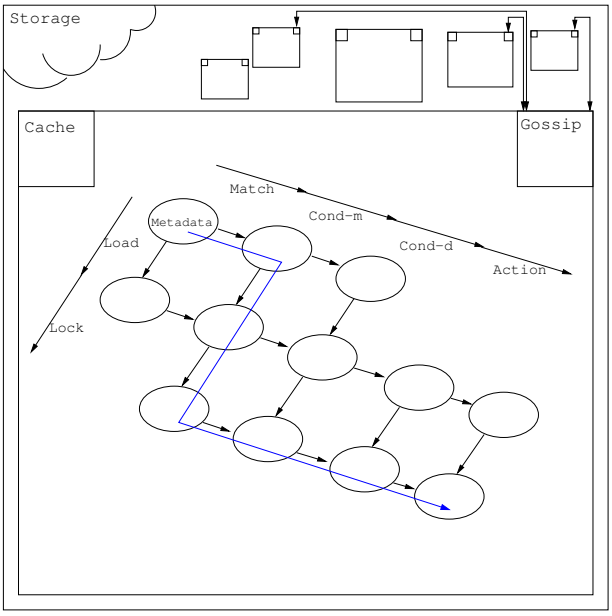

Figure 5: EarlyIO strategy.

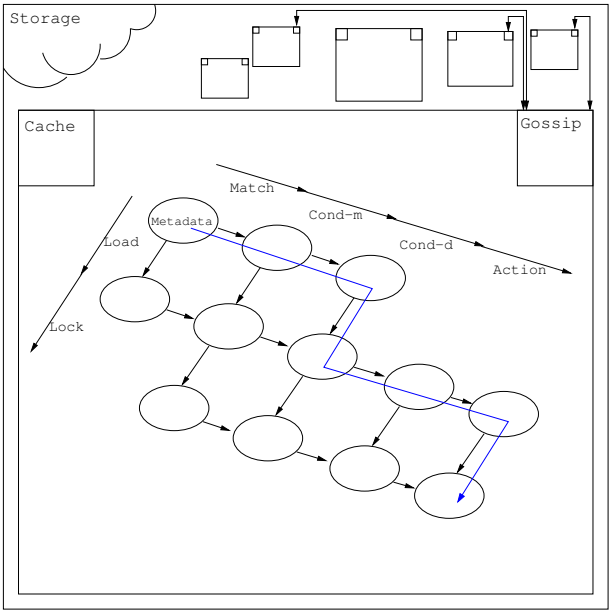

Figure 6: EarlyCPU strategy.

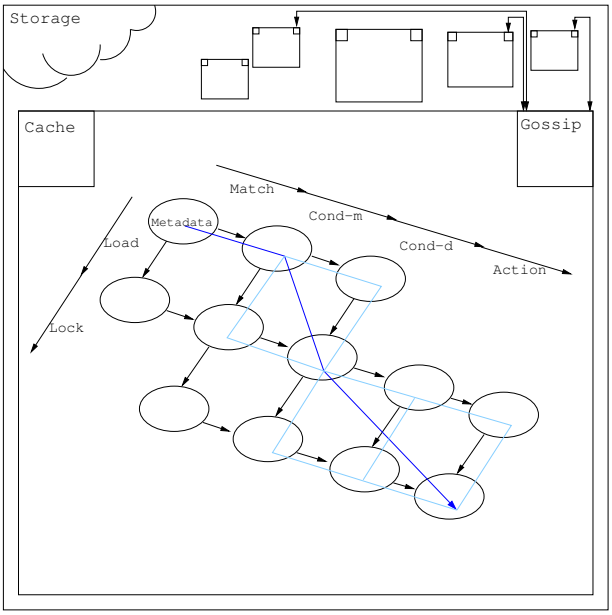

Figure 7: Parallel Strategy.

***Parallel*:** By mixing the execution of CPU and I/O tasks, this strategy tends to reduce the time spent. Whenever a match is found, both the Metadata test and Data is fetch are performed concurrently. Thus, the time spent performing these two tasks is lowered from $t_{test_m} + t_{load}$ to $max(t_{test_m}, t_{load})$ where t_{test_m} is the time the test on Metadata takes to complete and t_{load} is the time required for the Data to be loaded.

When both loading and testing tasks have returned, the remainder of the computation is executed, with concurrent CPU and I/O parts. Time spent here is also reduced.

Figure 7 shows the path used by this strategy.

3.4.3 Busy Wait

Another optimization that can be added once the CPU and I/O tasks are identified as orthogonal is busy wait. Whenever waiting for an I/O operation, a node can perform CPU tasks and, dually, when a long computation is carried out it is possible to use the idle network resources.

3.5 Cost Model

Considering the aforementioned strategies, one can wonder about which is the better one. In order to get a better understanding of the issue at hand, we

propose the following formalization.

Cost Model: Each — CPU or I/O — step is attributed a success rate s and a cost c. The success rate indicates the probability of a positive outcome for the execution of the associated step. The cost measures the time and resources spent in the execution. Using these numbers we can reduce the strategy selection to an optimization problem. The optimization problem can be treated in different ways. One is to minimize resource wastage. Another is to maximize the number of replacements per unit of time.

As is, cost estimation is difficult because it reflects both time — which we can try to minimize — and resource consumption — which may have a strict higher bound.

Dynamicity: Let us remember that the behavior of an application can change dynamically. Indeed, when performing replacements, new molecules can be introduced in the solution. Thus, potentially making new rules applicable. These new rules can have a different behavior. Moreover, when considering higher order, rules cannot be known before the execution. These concerns guided our choice toward a dynamic estimation of the reaction behavior and strategy adaptation. Estimating the behavior of the reaction can be done locally by gathering statistics about basic operations.

3.6 Envisioned Improvements

The internship was not long enough for us to pursue all the developments we envisioned for our Chemical Machine. The elements given here have not been studied thoroughly neither does the simulator detailed in Section 4 take them into account.

3.6.1 Load Balancing

There is no guarantee in our Chemical Machine that nodes perform the same amount of computations. Either the randomness of Metadata exchange can be shown to provide statistical guarantee or it is necessary to devise a load balancing scheme. Statistical guarantees would not apply when the number of possible reaction is low. Moreover a reaction rules can impose an arbitrary burden on the nodes, making it preferable to balance the load.

Let us define the potential of a multiset as:

Potential The potential $p_b(M)$ of a multiset M with respect to predicate b is equal to $\frac{Card(\{\vec{x} \in M^* | b(\vec{x})\})}{Card(M^*)}$ where M^* is the multiset of tuples of elements of the multiset M.

A node can dynamically estimate the potential of its metadata solution with respect to the match predicate. As demonstrated in [8] it is possible to use

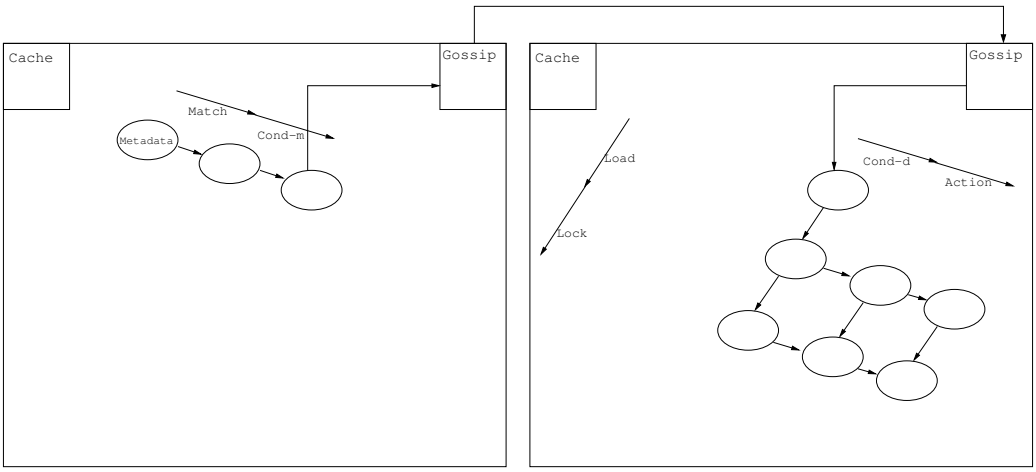

Figure 8: Load balancing.

gossip to evaluate the mean of a value on the whole P2P system[16]. Thus allowing a node to compare the global mean potential with its local value. Such a comparison can allow a node to decide if load balancing is necessary.

Detecting the appropriate time for balancing the load is not enough. We propose to carry the load balancing as follows:

- a node decides to arbitrarily interrupt its work;
- partial results are transmitted to a peer; and
- the receiving peer carries on the work to completion.

The interruption can happen in different steps of the computation. Depending on the lock system used in the machine, it might even be possible to share tasks using locked Data. Figure 8 illustrates the load balancing scheme.

3.6.2 Sifting Through Metadata

When looking for a match, there are several ways to sift through Metadata. In [9] several optimizations are given on top of naive backtracking. All these optimizations are geared toward a parallel usage. As our machine is distributed, it is not possible to apply all of them. Looking for a match is an important part of the Chemical Machine process and needs to be carefully designed.

[16] In [8], it is used to dynamically estimate the mean upload capability of peers in the system.

3.6.3 Specialized Nodes

Another possible improvement we thought of consists in specializing some nodes of the machine. Because homogeneity of nodes makes the system easier to think about and the code simpler to maintain, these changes are to be thought of as leads to be considered rather than perks that must be integrated. These special behaviors we propose for some nodes might also be diffused on the whole system so as to preserve homogeneity.

Some nodes can hoard large amounts of Metadata and spend all their computing resources finding matches. By not performing tests nor substitutions, these hoarders can treat a more important number of potential matches. These can help solve an issue associated to high-arity rules. For a n-ary rule to be applied, all the n appropriate molecules need to be located on the same node.

An other way to specialize nodes is to have Metadata re-generators. Those nodes' role is to make sure no molecule end up without its associated Metadata. Such a molecule would never react as it would not be accessible by the peers. Depending on the storage device implementation, the way re-generators would perform their task would vary greatly. A general idea is to induce decay on Metadata and have re-generators regularly introduce fresh values in the solution.

4 Simulator

Rather than developing a complete prototype for the Distributed Chemical Machine
described in Section 3, we created a simulator. This trade of has been decided primarily because of limited development time.

The simulator has been built from scratch. It has been considered interfacing with or extending an existing P2P simulator but we decided against it for the reasons explained thereafter.

Extending: As the Chemical Programing paradigm consist of rewriting rules applied on immutable values (molecules), Functional Programming seems far more suited than Imperative or Object Oriented[17] for simulation and implementation. Moreover, personal skills favored Functional Programming. Among the P2P simulator we came across and studied[18] were Java and C++ implementations. Thus extending an existing P2P simulator was rejected.

Interfacing: Interfacing with an existing simulator, with the P2P and Chemical parts treated by two distinct components, is technically possible. We thought about having two processes running; one simulating network and the other chemistry. When considering the pros and cons of this approach we stumbled upon the important scheduler issue. We found P2P simulators with two distinct approaches: preemptive threads and discrete time, cycle-based event loop. In the former, each node is associated to a kernel thread thus potentially benefiting from multiprocessor but suffering from preemptive scheduling issues: the need for bug prone mutexes and the number of concurrent threads limit. In the latter, each node can register events — such as sending a message — in a queue. Events are treated in order thus providing good performance, especially since there is no context switch. Because it is cycle based, nodes are forced to progress at a similar pace, with an important impact on realism.

Interfacing with preemptive threads is easy enough. However writing a dead-lock free preemptive threading program can be challenging and may need an important testing and debugging time. Interfacing with cycle based event loop is fairly easy if one writes with an identical scheduling style. Indeed, because the pace of distinct nodes is dictated by the cycle based scheduler, nodes are sometime forced to stop and wait. Using different schedulers for different components of the program can cause dead-locks — each component waiting on events from distinct nodes.

[17] The current HOCL implementation available in the MYRIADS team is written in Java [14]. As the semantic of HOCL does not include any notion of state nor environment, it is the programmer's responsibility to be careful with side effects.

[18] p2psim: `http://peersim.sourceforge.net`, PeerSim: `http://pdos.csail.mit.edu/p2psim` and simuGossip (ongoing development in the ASAP team, not released yet).

4.1 Technical Description

4.1.1 Concurrency

As the simulator multiplexes several nodes and a storage system, it is necessary to handle some level of concurrency. This is achieved through the use of a cooperative threading library called LWT[19] (for Light Weight Threads) which usage is detailed in Annexe A.1.

Cooperative threading is a technique in which each thread has to explicitly step down so as to allow others to run. This is achieved by the thread either sleeping, yielding or making a system call. An obvious downside is that a single non cooperative thread can ruin the whole program. This is balanced by the cheap creation and switching of threads which provide good performance and allow a much higher number of threads to run concurrently. Most of all, as cooperation points are explicit, mutexes are rarely needed: the critical section simply needs to be placed in a non-cooperating piece of code.

LWT provides a monadic interface with syntax extension which integrates well in the OCaml language. We present the following few lines of code extracted from the simulator — with some simplifications — as an example.

```
let use_locked_tested_data rule data state =
  lwt product = Chemistry.action rule data in
(*^^^ lwt is a cooperative binding operator*)
  let actions = Bag.add state.actions (Locked data, product) in
(*^^^ let is a non-cooperative binding operator*)
  return {state with actions;}
(*^^^^^^ the monad's return function*)
```

The function `use_locked_tested_data` computes the substitution performed by a given reaction `rule` on a given set of `data`. It is assumed that the `data` value has successfully passed the rule's test and that a lock is owned. The first statement binds the result of the set of molecules produced by the reaction rule to `product`. This binding uses the cooperative construct `lwt`, meaning that the computation takes time to complete and that other threads may run concurrently. The second statement adds the previous result to `actions`, the set that holds the computed substitutions. The last line returns an updated `state` where the `actions` field is modified.

4.1.2 Code organization

As a software engineering principle, the simulator makes heavy use of modules and functors. Functors are to modules what functions are to values. Not only does it make code generic and easier to maintain, it also helps isolating the different nodes. Figure 9 gives a partial dependency graph of the most important modules a node is made of.

[19] `http://ocsigen.org/lwt`

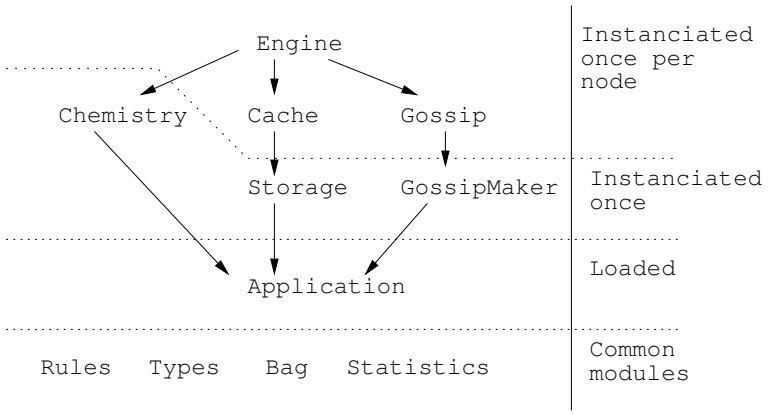

Figure 9: Important module dependencies at the node level.

The `Dynlinker` module (note represented in Figure 9) is responsible for loading the Chemical application into the simulator. This is achieved through the use of OCaml's dynamic linking feature. When loaded, the application registers an `Application` module that is used to instantiate specialized modules. Nodes don't have access to the `Dynlinker` module which only role is to instantiate new peers.

Among the specialized modules the one called `Storage` simulates the storage device discussed in Section 3.2. This module is instantiated via a functor taking the application as argument. This functor application provides the storage device with the initial solution.

Each node is created by using the dynamically loaded `Application` module as argument to several functors. It makes node isolation possible as module specific variables cannot be tampered with by other nodes. In our implementation, the `Chemistry` module, responsible for the CPU steps, does not use global variables. Each node can safely use the same module with no possible interference. Functor instantiation is only needed once so as to transmit the Chemical rules.

4.1.3 Working with Functional Programming

Handling Arbitrary Arity: As OCaml is a strongly typed language the arbitrary arity of rules had to be handled. Indeed, a rule can accept tuples of arbitrary size. Several solutions were considered:

- using a variant type;
- using lists — or arrays — as variable size tuples; or
- using continuations as variable arity functions

The variant type approach is quite straightforward but imposes an upper bound on the rules arity as well as an important amount of boilerplate code.

We chose continuations over lists/arrays because we feel it is easier to handle the matching process. Indeed, when looking for a match, we perform matching attempts on incomplete tuples of molecules. Handling these attempts with continuations is quite concise. The type used in the continuation based solution is detailed thereafter. In the match operation it is instantiated with the type of Metadata for `'argument` and `bool` for `'result`.

```
(** Arbitrary arity constructs *)
type ('argument, 'result) anyary =
  (** At least another argument is required *)
  | More of ('argument, 'result) plusary
  (** The computation is done *)
  | Done of 'result

(** Non-null arity construct *)
and ('argument, 'result) plusary =
  'argument -> ('argument, 'result) anyary
```

Leveraging Value Sharing: A very useful feature of most Functional Languages is *value sharing*. It allows one to refer to a value in two different context without the need for it to be duplicated in memory. Its representation is shared without raising any coherence issues as values are immutable.

This is useful in our simulator as Metadata can be used by several nodes. Data is also used concurrently in different places. Thus, value sharing helps keeping the memory usage low and reduces hardware requirements associated with running the simulator.

4.2 Simulation Abstractions and Limitations

A simulator always abstracts some aspects of the problem it helps studying. E.g. simulating the whole TCP/IP stack and interference in wires is not useful when trying to evaluate the failure resilience of a distributed application. Similarly, in our case, elements of the system are made simpler. We here list them.

4.2.1 P2P Model

It has been shown [10] that RPS can achieve true randomness. The two sole P2P constructs that are used in our machine are RPS and messaging. For this reason, the P2P aspects of the simulation are drastically abstracted.

The RPS is achieved by having nodes registering themselves in a global variable[20] and randomly picking an element from this set when needed. This is a

[20]The variable is instantiated during initialization, after the application module has been loaded.

debatable choice. However, as we do not account for node failure, simulating a complete, full fledged gossip simulator seems excessive in this early stage of development.

Each node has an associated stream of incoming messages which can be written to by any of its peers. The use of functor described in Section 4.1.2 ensures that a mailbox can only be read by the peer owning it. This is achieved by having the mailbox created on node instantiation and only registering the writing function in the global peer registry.

4.2.2 Inertia Detection

A feature that was not implemented in our simulator is inertia detection. Detecting stable state makes it possible to stop the Chemical Machine when no rule can further be applied. When no rule can be applied, seeking matches is a waste of resources. While necessary if one is to perform an algorithmic computation — meaning *a computation on a Turing Machine* —, let us remember that it is not all that — so called — computers do. Indeed daemons and servers do not fit in the description of a computation as they never cease to run, always waiting for possible, unpredictable input.

Implementing inertia detection in a Chemical Machine is not trivial. All the possible combinations of molecules have to be tested[21]. In a distributed setting, especially using a loosely synchronized, gossip powered overlay, this task is even harder. It is not sufficient to detect inertia on all the nodes. While we chose not to implement it, it is not impossible to do. An easy solution is to:

- pair stable nodes together,
- move all the molecules of the pair to one of its node,
- stop the empty node.

This tends to aggregate all the molecules to a single node, which is an appropriate course of action for certain reactions. If the number of molecule decreases, then merging stable nodes can be an efficient strategy.

Because detecting inertia is not necessary and because its development is time consuming, the implemented simulator does not provide it.

4.3 Tweak-able Parameters

The simulator seems rather limited given its limitations. We here list the simulation parameters that can be changed so as to provide interesting results.

The peer messaging system is type safe as the type of messages is parametrized by the application specific Metadata and Data types.

[21] It is possible that a single test takes out multiple combinations as explained in Section 2.2. Moreover, if the arity n of rules is known, it is not necessary to test m-tuples if $m \neq n$. This can help to temper the issue, but does not solve it.

4.3.1 Application Specific Parameters

As mentioned before, an application is dynamically loaded in the simulator. The loaded module contain application specific characteristic such as:

- the set of rules and the initial set of molecule the reaction has;
- the expected delays for Data and Metadata transmission; and
- the choice of the strategy being used.

The rules and initial molecules are not parameters of the machine per se. It is nevertheless important, for simulation purpose, to be able to test different applications as they can expose different behaviors.

Network delays are faked through the use of a sleep-like cooperative function that make a thread inactive for a given amount of time. The delays for Data and Metadata can be specified in the application module. Delays for messages that do not carry Data nor Metadata are considered infinitesimal — the thread is pushed to the end of the scheduler queue but does not become inactive.

The chosen strategy is regularly re-evaluated for each node. This evaluation uses statistics about the current state of the node and about the state other peers are in. The former is a count of the number of operations performed along with their success rate. The latter is estimated through the exchange of messages carrying the aforementioned count.

4.3.2 P2P Parameters

By successively instantiating the needed functors, the simulator can create several nodes. The number of those can be specified using a command line argument.

4.3.3 Nodes Parameters

All the nodes of our machine are configured similarly. They all share similar parameter values. These are listed here.

Super Cycles: Every once in a while, each node re-evaluates its strategy, gossips measured statistics around and swaps a batch of molecules with a peer. The period it uses for doing all this is called *super cycle*. A super cycle happens after the node performed a given number of operations. This number can be specified in the command line.

Cache Size: Another parameter the nodes can be modified with is the cache size: the number of molecules that the cache module can hold.

Note that we use a match seeking technique that makes cache less useful than with a backtrack algorithm. The Metadata sifting algorithm was discussed in Section 3.6.2. In our method the mean time between to successive test of a molecule is high.

Molecules per Node: Each node tries to maintain an equilibrium between too much molecules and too few. An overwhelming amount of molecule creates a local combinatorial explosion. On the other hand having a handful of molecule available makes it quite quick to match all the possible combinations of molecules. Neither situation is suitable. The former calls for more nodes — more computing power — with less molecules per node and the latter makes it necessary to swap molecules more frequently.

4.3.4 Simulator Parameters

Some parameters impact only the simulator and not the running simulation[22]. This include verbosity of the output: the simulator can be totally muted or made to output a lot of information. A highly verbose output is useful mainly for debugging the simulator or the Chemical application.

The simulator can also output reports for the different statistics a node collects. Each node writes its own report to a file in a gnuplot readable format. These are used to generate charts such as those presented in Section 5. The simulator can be told to deactivate activity reporting with a command line argument.

[22]While these parameters do not drastically impact the core of the simulation, the induced change in the I/O activity level can induce slight differences. The core of the LWT library is deterministic, but the use of I/O primitives is not.

5 Simulations and Results

We here present the effect of two distinct strategies — namely *earlyIO* and *earlyCPU* as presented in Section 3.4.2 — on two distinct programs with different behaviors. Both programs are abstract Chemical reactions modeling different kind of real life applications.

The first program has tautological tests and takes a lot of time to compute the product of the reaction. This program is said to be computationally intensive. The second application is its dual: low success rate, lengthy tests and trivial substitutions. Another difference between the two applications is the network delays used. A higher simulated latency is used in the second application, thus making network operations longer. For this reason, the second application is said to be data intensive.

The computationally intensive application models a particular usage where molecules are simple jobs to be performed by the nodes. The data intensive application share similarities with real life applications such as changing the metadata of a collection of photos.

We ran the two applications during 150 seconds with a total of 5 peers working each on 100 molecules. Both applications use an initial solution of 500 molecules which decreases as the reaction progresses.

We compared results using the statistics gathered by a random node[23] within each Chemical Machine. All the nodes in any given simulation presented similar patterns with slight numerical variations.

5.1 Computationally Intensive Application

In this application, there is only one reaction rule. It takes 2 seconds to complete the substitution and tests are tautological.

The chart presented in Figure 10 exposes the total number of lock attempts and successes for both *EarlyIO* and *EarlyCPU* strategies for the computationally intensive application. There is a remarkable attribute, the chart makes visible: the *EarlyCPU* strategy progresses slower than its counterpart. Indeed, statistics are outputted every super cycle. *EarlyIO* performs 17 of them in the 150 seconds of the simulation while *EarlyCPU* only has 10. This is due to the fact that *EarlyCPU* aborts operation only after executing the long substitution computation while *EarlyIO* aborts before — thus avoiding time wastage. Lock failure rate difference is slight.

When comparing the number of replacement performed it is clear that the *EarlyIO* perform better than its dual strategy as exposed in Figure 11. Let us remember that the results shown here are not global to the Chemical Machine, but specific to a node. Every peer in the Chemical Machine makes it progress at a similar speed.

[23] In each case the node with the lowest ID was observed. This ID is generated using OCaml's Random module and used only for logging and statistic reporting purpose, thus not interfering with the simulation.

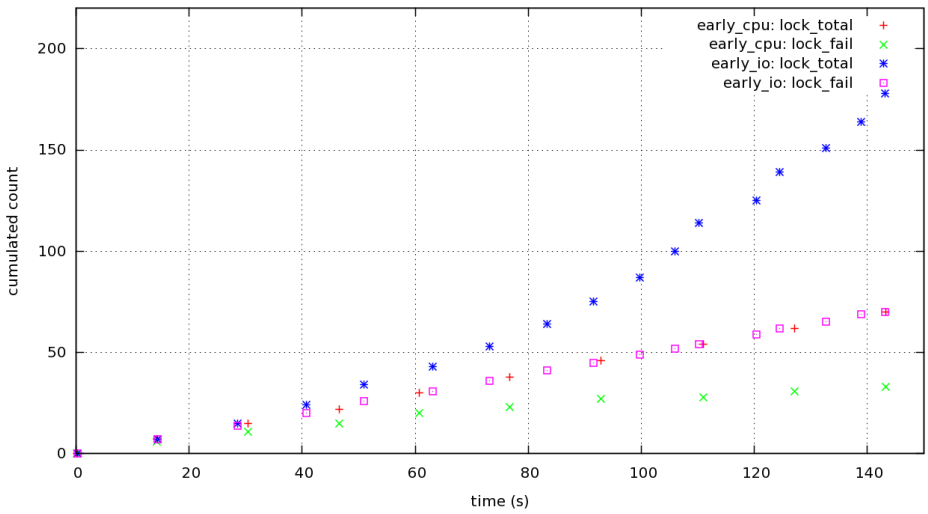

Figure 10: Lock operations in the computationally intensive application.

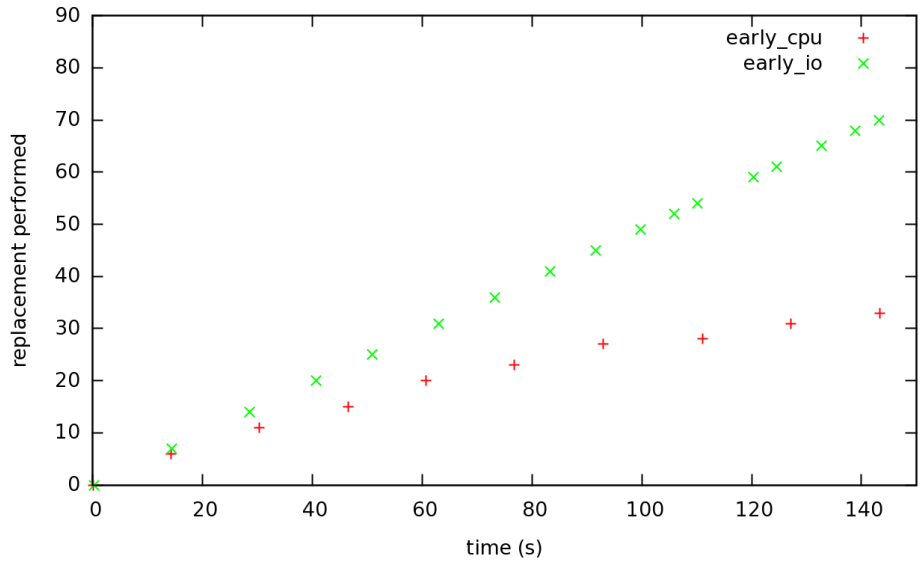

Figure 11: Replacement operations in the computationally intensive application.

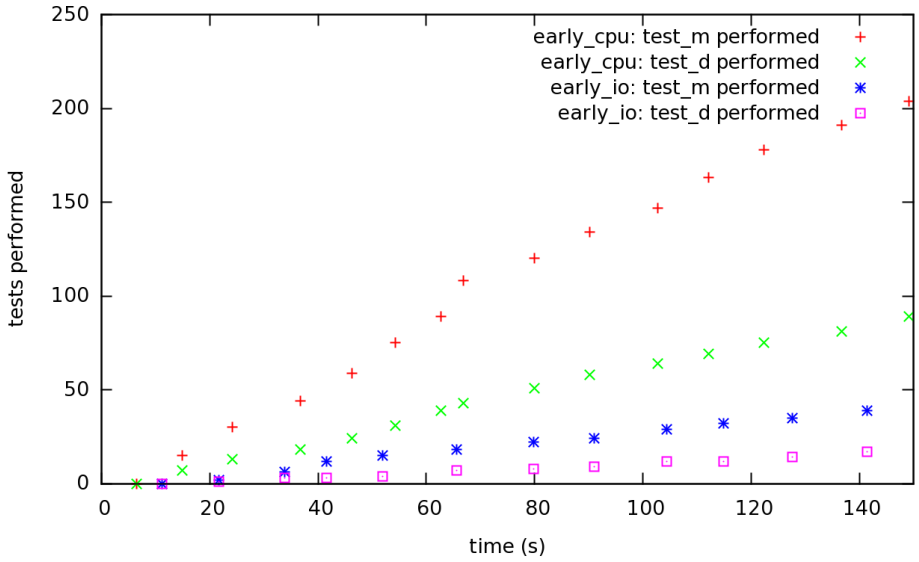

Figure 12: Test operations in the data intensive application.

5.2 Data Intensive Application

This application also has a single rule. The tests have an approximate 50 percent chance of failure — each. One important difference with the computationally intensive application is that the simulated network delays are higher. Hence the cost — in number of seconds — of network operations is higher.

Figure 12 shows the cumulated number of tests — both on Metadata and Data — performed by the two different strategies. Unlike the dual application, we can see here that *EarlyCPU* performs better with 15 super cycles instead of 12 for *EarlyIO*. We can also see that each cycle performs more tests with the *EarlyCPU* strategy. This can be explained by the fact that the network operations are costly. As *EarlyCPU* tends to reduce their number by performing the potentially failing tests before, it is not as impacted as *EarlyIO* by this increased cost.

Comparing the number of replacements each strategy can perform in the observed time interval, *EarlyCPU* appears to be better suited for the given application. This fact, shown in Figure 13, confirms our expectations.

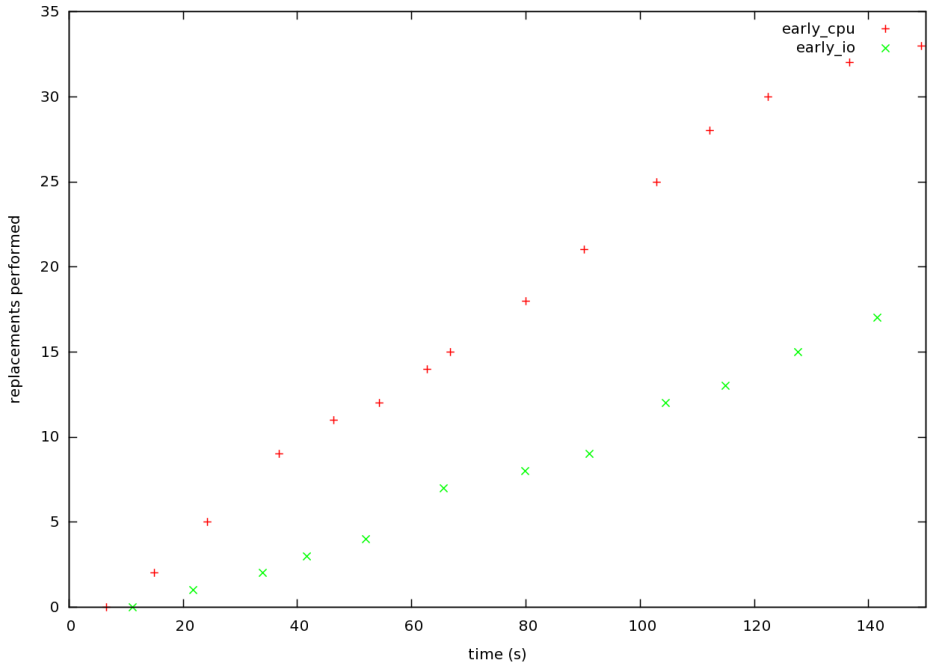

Figure 13: Replacement operations in the data intensive application.

6 Conclusion

Joint Work: The internship is the result of a joint work with the MYRIADS[24] and ASAP[25] teams. MYRIADS focuses on autonomous service management in distributed architecture and ASAP works on large-scale, server-less, distributed systems. Both concerns are related to the internship as the aim is to study and build a distributed, P2P system geared toward the execution of Web Service composition. Both Marin Bertier and Achour Mostefaoui, members of the ASAP project, took part in discussions leading to design choices on the Chemical Machine. This collaboration is not restricted to the internship.

Related Projects: The two teams are currently working on a couple of related projects. A DHT with support for atomic retrieval of multiple elements is being worked on. It could be used to implement the storage device in a distributed fashion. Such a DHT allow one to perform an operation on several elements atomically, thus making atomic capture possible.

During the internship, we investigated a gossip-based model with random execution pattern. As part of its Ph.D. thesis, Marko Obrovac, works on an alternative solution based on a distributed divide and conquer, where solutions are split, independently brought to inertia and merged. The result is then itself reduced to inertia. A perk this solution offers is the possibility to leverage the inertia of sub-solutions: when merged, reactions can only happen if at least one element in each sub-solution is used. A mixed reduction strategy with ideas drawn from both the gossip and divide-and-conquer solutions makes up an interesting research topic.

Contribution: In the first four months of this internship, a completely distributed model for a Chemical Programing paradigm runtime has been devised. This Chemical Machine has been partially formalized, with a particular focus on the strategies.

A simulator was implemented from scratch for this Chemical Machine. It allowed the test and validation of the importance of the choice of a strategy.

Future Work: Formalization will have to be completed. A proof of the randomness of the scheduler is important, especially if the extensions mentioned in Section 3.6 are to be used. Alternatively, the machine could be adapted so as to provide stronger guarantees for its scheduler.

Another area where the formalization can turn out to be important is when mapping application behavior to strategies. As explained in Section 3.5 there exists several measurements of the efficiency of a strategy. More time needs to be spent on this topic.

The storage device is abstract — much like oracles in computability problems. In order for the machine to be usable, it is necessary to make it real.

[24] `http://www.irisa.fr/myriads`
[25] `http://www.irisa.fr/asap`

Aforementioned work about the atomic DHT is a promising lead but does not solve the problem entirely. Integrating the two systems into one is necessary.

The simulator also needs to be worked upon. Several limitations and implementation choices can be lifted and made customizable. The load balancing scheme detailed in Section 3.6.1 could be tested in our simulator. The P2P overlay needs a more realistic simulation. Different methods could be experimented with in order to sift through the Metadata. The simulator is meant to evolve.

The model itself needs to be worked on. Important features such as failure resilience and self-adaptation are necessary for the Chemical Machine to run in hostile environments such as the Internet.

The issues raised during the internship cannot be solved in the time that remains. Short term goals are to improve both the formalization and simulator. Making extra features available is left for another internship or a Ph.D.

A Annexes

A.1 The LWT Library

As mentioned in Section 4.1.1, the simulator uses the LWT library to handle concurrency. LWT provides a monadic interface detailed thereafter. Functions that may not return immediately carry the monad type in their signature making cooperation points obvious and mutexes unnecessary. Moreover threads are first class values which eases their manipulation — e.g. cancellation, synchronization. The monad, along with elementary manipulation functions are presented in the following interface:

```
type 'a t (* The type of cooperative threads returning 'a values *)
val return: 'a -> 'a t
val bind: 'a t -> ('a -> 'b t) -> 'b t
val cancel : 'a Lwt.t -> unit
val join : unit Lwt.t list -> unit Lwt.t
```

Explicit cooperation can be achieved through the use of `Lwt_unix.yield` which moves the current thread to the end of the scheduler queue or `Lwt_unix.sleep` which makes the thread returns after a given number of seconds. All blocking system calls are also wrapped in a proper non-blocking, cooperating function. When a thread start sleeping — either by explicit cooperation or system call — the scheduler picks another thread that is ready and executes it.

Cooperative scheduling uses the concept of threads when it comes to workflow — with a special continuation passing style in the case of LWT — but is similar to event loop scheduling internally. Indeed a thread that cooperates simply registers an event with the appropriate continuation in the scheduler queue.

A syntax extension further helps the integration of the library. Using bind operation makes the use of a monad obvious. On the other hand, the extended syntax is quite similar to the original as demonstrated in the following example:

```
(* using bind *)
bind
  (read_char input_channel)
  (fun c -> write_char output_channel c)

(* using the ( >>= ) infix operator *)
read_char input_channel >>= fun c ->
write_char output_channel c

(* using the extended syntax *)
lwt c = read_char input_channel in
write_char output_channel
```

η-conversion can arguably make the bind function and operator based solutions more elegant. However, this is not always possible.

Complete documentation for the library along with the sources can be found at `http://ocsigen.org/lwt/` and formal description along with scientific considerations can be found in [16].

A.2 Chemical Expression of Web Service Composition

While it was not studied in the internship, understanding how Web Service composition can be expressed in the Chemical Programing paradigm is useful to understand the context of this work. Complete study of Chemical workflow specification can be found in [7].

One of the possible approaches is to use requests as molecules and Web Services as rules[26]. Hence, whenever a client makes a request, a new molecule is introduced in the solution. This molecule will react with the requested Web Service so as to produce a result — that will be sent back to the client — or a collection of intermediate results that will trigger the execution of other Web Services. A Web Service can consume several intermediate results simultaneously.

As an example, consider a Web Service *WS* providing a search mash-up: a user inputs a query and result from several search engines (*S1*, ..., *Sn*) are presented together. The user makes a request which introduces a molecule in the solution. As soon as this molecule meets the rule responsible for *WS* it is substituted with the Data necessary to the execution of the requests on the search engines *S1* to *Sn*. Another molecule is inserted in the solution, one that makes it possible to gather and merge all the results together. If considering higher level Chemical Programing this molecule can be a rule. Here is an approximate solution:

```
WS = replace q::Query
     with q1::RequestS1,
          ...,
          qn::RequestSn,
          (replace r1::ReplyS1,
                   ...,
                   rn::ReplySn
           with merge(r1, ..., rn)
          )
```

The syntax used is HOCL's, with the exception of `...` which is not a valid language construct.

The presented rule is a poor approximation whose only aim is to give a general idea of how Web Services can be programmed using Chemical Programing. Among the problems this implementation has is a race condition when several requests are treated simultaneously.

[26] As seen in Section 2.1.2, Chemical Programing can be higher order. This can blur the distinction.

Other methods can be created. As this is out of the scope of this work, we do not dwell on them here.

A.3 Multiset

As stated in Section 2.1.2, a multiset resembles a set with the notable difference that elements can appear more than once. The number of occurrences of an element is called its *multiplicity*. The set of all the elements of a multiset is called its *support set*. Hence, a multiset can be thought of as a function associating to each and every element of the support set, its multiplicity — a function from the support set to $\mathbb{N}$.

Multiset A Multiset of support S is a function from S to $\mathbb{N}$.

Using this formalism, it becomes easy to extend the well known usual operations that exists for the sets in order to make them available on multisets. Let A and B be two multisets of support S. We define:

$+$ **(merge)** $\forall e \in S, (A + B)(e) = A(e) + B(e)$

$-$ **(remove)** $\forall e \in S, (A - B)(e) = max(A(e) - B(e), 0)$

$\cap$ **(intersection)** $\forall e \in S, (A \cap B)(e) = min(A(e), B(e))$

$\cup$ **(union)** $\forall e \in S, (A \cup B)(e) = max(A(e), B(e))$

Note that both merge and union operations are extensions of the union of sets. This is due to $\mathbb{N}$ having a richer structure than $\{0, 1\}$. Also note that we do not consider negative multiplicities — hence the max in remove's definition. As mentioned in Section 2.1.2 some extensions to the notion of multiplicity are described in [14].

Predicates on multisets can be easily define via lifting — e.g. inclusion on multisets ($\subset$) is the lifting of less-than on integers ($<$).

$\hat{}$ **(lift)** let P be a n-ary predicate on integers, we note $\hat{P}$ the n-ary predicate on multisets such as $\forall$ support set S, $\forall A_1, \ldots, A_n$ multisets of support S, $\hat{P}(A_1, \ldots, A_n) \equiv \forall e \in S, P(A_1(e), \ldots, A_n(e))$

B References

References

[1] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Comput. Surv.*, 36(4):335–371, 2004.

[2] Gheorghe Păun. Computing with Membranes. *Journal of Computer and System Sciences*, 61:108–143, 1998.

[3] Jean-Pierre Banâtre, Anne Coutant, and Daniel Le Métayer. Parallel Machines for Multiset Transformation And their Programming Style. *INRIA, Rapport de Recherche*, November 1987.

[4] Gérard Berry and Gérard Boudol. The Chemical Abstract Machine. In *POPL*, pages 81–94, 1990.

[5] Marin Bertier, Yann Busnel, and Anne-Marie Kermarrec. Dynamic Computation of Population Protocols. *IEEE International Conference on Telecommunications ICT2010*, May 2010.

[6] Bittorrent.org. `http://www.bittorrent.org/beps/bep_0003.html`.

[7] Héctor Fernández, Thierry Priol, and Cédric Tedeschi. Decentralized Approach for Execution of Composite Web Services Using the Chemical Paradigm. In *ICWS*, pages 139–146. IEEE Computer Society, 2010.

[8] Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, Boris Koldehofe, Martin Mogensen, Maxime Monod, and Vivien Quéma. Heterogeneous Gossip. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 3:1–3:20, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[9] Katia Gladitz and Herbert Kuchen. Shared Memory Implementation of the Gamma-Operation. *J. Symb. Comput.*, 21(4):577–591, 1996.

[10] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-Based Peer Sampling. *ACM Trans. Comput. Syst.*, 25(3), 2007.

[11] Craig Labovitz, Scott Iekel-Johnson, Danny McPherson, Jon Oberheide, and Farnam Jahanian. Internet Inter-Domain Traffic. In Shivkumar Kalyanaraman, Venkata N. Padmanabhan, K. K. Ramakrishnan, Rajeev Shorey, and Geoffrey M. Voelker, editors, *SIGCOMM*, pages 75–86. ACM, 2010.

[12] OASIS-consortium. BPEL. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`.

[13] OASIS-consortium. WS. http://www.oasis-open.org/committees/tc_cat.php?cat=ws.

[14] Yann Radenac. *Programmation chimique d'ordre supérieure*. PhD thesis, Université Rennes 1, 2007.

[15] Michael Rosen, Boris Lublinsky, Kevin T. Smith, and Marc J. Balcer. *Applied SOA*. John Wiley and Sons, Inc., 2010.

[16] Jérôme Vouillon. Lwt: a Cooperative Thread Library. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, ML '08, pages 3–12, New York, NY, USA, 2008. ACM.

[17] W3C. HTML Living Standard. http://www.whatwg.org/specs/web-apps/current-work/multipage/webappapis.html#webappapis.

[18] Wired. The Web Is Dead. Long Live the Internet. http://www.wired.com/magazine/2010/08/ff_webrip/all/1.