Distributed Computations Driven by Resource Consumption

Luc Moreau
Department of Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ, UK
L.Moreau@ecs.soton.ac.uk

Christian Queinnec
LIP 6 & INRIA-Rocquencourt
4, place Jussieu, 75252 Paris Cedex France
Christian.Queinnec@lip6.fr

Abstract

Millions of computers are now connected together by the Internet. At a fast pace, applications are taking advantage of these new capabilities, and are becoming parallel and distributed, e.g., applets on the WWW or agent technology. As we live in a world with finite resources, an important challenge is to be able to control computations in such an environment. For instance, a user might like to suspend a computation because another one seems to be more promising. In this paper, we present a paradigm that allows the programmer to monitor and control computations, whether parallel or distributed, by mastering their resource consumption. We describe an implementation on top of the thread library PPCR and the message-passing library Nexus.

1 Introduction

As we live in a world with finite resources, it is of paramount importance for the user to be able to monitor and control computations. This task is all the more complex since computations may be parallel, distributed, and most probably make use of code written by others. This problem is particularly illustrated by the Internet as the user wishes to search information over the WWW, exploits parallelism to improve efficiency, relies on distribution to increase locality, but also wants to concentrate the computing power in the most promising directions to reduce searching time.

There are two types of applications that we particularly wish to program. First, a user that has initiated a computation over the Internet should be able to suspend the computation in order to analyse the results he has already obtained. If these were unsatisfactory, s/he must be able to resume the computation from the point where it was suspended in order to collect the next results. In this example, computations should be understood as possibly parallel and distributed. Second, service providers offer computing facilities or resources to customers who subscribe to their service by transferring electronic cash [38]; in return, service providers supply them with a handle to submit jobs, create accounts that they debit according to the usage of the facilities, and inform users of the exhaustion of their account. Again, jobs submitted by customers may generate parallel and distributed computations which must be monitored by the service provider.

Our goal is to provide the means by which everybody, customers and service providers, can get the most anyone can out of a situation with bounded resources. We believe that parallel computations can be driven by mastering their resource consumption. In this paper, we present a new paradigm, called Quantum, that allows the user to monitor and control distributed computations. The basic principle of Quantum is to keep track of the resources consumed by computations. Resources can be understood as processors cycles, bandwidth and duration of communications, or even printer paper. We shall adopt a more generic view by saying that computations need energy to be performed\(^1\). Although the notion of energy is part of Quantum, the programmer cannot create energy ex nihilo, but can only transfer it between computations via some primitives. As a result, we were able to ensure a general principle for Quantum: given a finite amount of energy, any computation is finite.

Quantum generalises some approaches adopted in agent scripting languages to control resources [27, 40]. Besides its resource-oriented foundations, Quantum is designed for parallelism and distribution. Furthermore, Quantum was conceived to be independent of any programming language, and in particular, of the primitives for parallelism and distribution, and of the memory model (central, distributed, with or without coherence). In order to prove this claim, we have built a library that implements the primitives of Quantum on top of the thread library PPCR [52] and the message-passing library Nexus [13]. Quantum is also integrated with Nexeme [32], a distributed implementation of Scheme [47]. Internally, Quantum energy is modelled by counters that are decremented every time a thread is scheduled; asynchronous notifications inform the user of energy exhaustion and computation termination. Two other primitives are able to supply or remove energy.

This paper is organised of follows. We present the intuition of Quantum in Section 2. In Section 3, we focus on the implementation of Quantum. Finally, Section 4 discusses related work and is followed by a conclusion. A companion paper [33] describes the formal semantics of Quantum and present several examples using its primitives.

2 The Library Quantum: Rationale

In this section, we introduce Quantum, its primitives and their intuitive semantics, and the considerations that lead to its design. The abstract syntax of Quantum primitives is displayed in Figure 1.

Quantum is independent of the primitives for parallelism or distribution. Parallel threads of evaluation may be created using Posix threads [22], or higher-level constructs like pcall \([34, 45]\) or future \([18, 31]\). In the sequel, we use the term task to denote an evaluation thread created by the constructs for parallelism. Similarly, message-passing style primitives \([12, 14, 13]\) or higher-level forms of remote function calls \([30, 32, 42]\) may be adopted for distribution.

\(^1\)Other names found in the literature for a similar concept are fuel \([20]\), computron \([46, \text{p. } 102–103]\), teleclick \([27]\), or metapill \([1]\).
Our goal is to be able to allocate resources to computations, and to monitor and to control their use as evaluations proceed. In our view, it is essential to be notified of the termination of a computation so that, for instance, unconsumed resources can be transferred to a more suitable computation. Similarly, we want to be informed of the exhaustion of the resources allocated to a computation, so that for instance more resources can be supplied.

In order to be notified of the termination or energy exhaustion of a computation, we need an entity that represents the computation. A group is an object that can be used to refer to a computation in a Quantum program. A group is associated with a computation composed of several tasks proceeding in parallel; in turn, they can initiate subcomputations by creating subgroups. As a result, our computation model is hierarchical. A group is said to sponsor [26, 37, 18] the computation it is associated with. Recursively, every computation has a sponsoring group, and so does every task.

At creation time, a group is given an energy quota. More specifically, a computation that evaluates the expression

\[ \text{call-with-group}(F,e \varphi_1 \varphi_2 \varphi_3) \]

under the sponsorship of a group \( g_1 \), creates a new first-class group \( g_2 \) that is allocated an initial quota of energy \( e \) and whose parent is \( g_1 \). Furthermore, it initiates a computation under the sponsorship of \( g_2 \) by calling \( F \) with \( g_2 \) and \( e \) as argument; hence, the user function \( F \) receives a handle on its sponsoring group. As Quantum keeps track of resource consumption, the energy \( e \) allocated to \( g_2 \) is deducted from the energy of \( g_1 \). Figure 2 displays the behaviour of the primitive call-with-group. We see a configuration where a group \( g_1 \) is sponsoring two tasks \( t_1 \) and \( t_2 \) and a subgroup \( g_2 \) itself sponsoring a task \( t_3 \). After evaluating the primitive call-with-group, a new subgroup \( g_2 \) sponsoring the application of \( F \) on \( g_2 \) and \( e \) is created; energy is transferred from \( g_1 \) to \( g_2 \). This transition assumes that \( e_1 > e \).

Remark In Quantum, every action has an associated cost. Figure 2 shows that the energy of \( g_1 \) is \( e_1 = e - 1 \) after transition. The value \(-1\) is the amount of energy transferred to the new group \( g_2 \) and \(-1\) represents the cost of the group creation operation. In order to simplify the presentation, we assume that all transitions have a unitary administrative cost.

Quantum enforces the following principle: any computation consumes energy from its sponsoring group. Therefore, not only is a group perceived as a way of naming computations, but also it must be regarded as an energy tank for the computation. Figure 3 displays a task \( f \) evolving to state \( t' \) by performing an action, whose cost \( e_1 \) is charged to the sponsoring group \( g_1 \).

In addition, two events may be signalled during the lifetime of a group: group termination and energy exhaustion are asynchronously notified by applying the user functions (the notifiers) \( \varphi_1 \) and \( \varphi_2 \), respectively. A group is said to be terminated, when it has no subgroup and it does not sponsor any task; i.e. no more activity can be performed in the group. In Figure 4, when the only task \( t_4 \) of group \( g_2 \) is terminating, the function \( \varphi_2 \) is asynchronously called with \( g_2 \) as argument to notify its termination, and the energy surplus of \( g_2 \) is transferred back to \( g_1 \). Note that the execution of the notifier \( \varphi_2 \) is sponsored by \( g_1 \), i.e. the parent of \( g_2 \).

In Figure 5, a computation \( t_2 \) sponsored by \( g_2 \) requires more energy than available in \( g_2 \); the function \( \varphi_2 \) is asynchronously called on \( g_2 \) to notify its energy exhaustion, also under the sponsorship of \( g_1 \), with transfer of the remaining energy of \( g_2 \) to \( g_1 \).

Remark An exhaustion notification, like every Quantum transition, has a cost. In order to guarantee that there is enough energy to notify any occurring exhaustion, we define the "exhaustion threshold" as the cost of notifying an exhaustion. An exhaustion notification will be raised if the cost of the current operation is higher than the remaining energy in its sponsoring group minus the cost of notification.

Figure 6 displays the state transition diagram for groups. At creation time, a group is in the running state, which means that the tasks that it sponsors can proceed as long as they do not require more energy than available. Asynchronous notifications are represented.

---

2 Subscript \( t \) denotes termination, whereas subscript \( e \) denotes exhaustion.
Energy may be caused to flow between groups, independently of the group hierarchy, under the control of the user program. Two primitives operate on groups: pause and awake. Intuitively, the primitive pause forces a running group and its subgroups into the exhausted state, and all the energy that was available in this hierarchy is transferred to the group that sponsored the pause action. The construct awake\((g,e)\) transfers energy \(e\) to the group \(g\), after deducting it from the group sponsoring the awake action. If the group \(g\) is in the exhausted state, its state is changed to running; if the group is in the terminated state, awake acts as a null operation.

Figure 7 displays the behaviour of awake, assuming that \(e_1 > e\) and \(g_2\) is not terminated.

Let us observe the non-symmetric behaviours of pause and awake.
that able to compute in a distributed framework. Hence, we decided
The single space parallel model is rather simple. Although tasks
3.1 Single Space Model
This section outlines the current implementation. First, it presents
Our permanent concern when designing Quantum was to be
3 Implementation Description
This section outlines the current implementation. First, it presents
3.1 Single Space Model
The single space parallel model is rather simple. Although tasks
awake: the former operates recursively on a group hierarchy, while
awake: the former operates recursively on a group hierarchy, while
awake: the former operates recursively on a group hierarchy, while
awake: the former operates recursively on a group hierarchy, while
awake: the former operates recursively on a group hierarchy, while
awake: the former operates recursively on a group hierarchy, while
with a given amount of energy, we remove this energy from the tank
Pausing a group $g$ is the most complex operation since it re-
cursively dries the tanks of all the groups in the tree rooted at $g$.
When $\text{pause}$ empties the tank of a group $g'$, it posts a notification
in the parent group of $g'$ mentioning the amount of stolen energy;
the notifier is the function given as second argument to $\text{pause}$; the
notification is a task whose job is to invoke the notifier with the
paused group and the amount of stolen energy.
Transitions described in Figures 2, 3, 4, 5 and 7 are to be
considered as atomic. On the other hand, the $\text{pause}$ function is not
atomic. The $\text{pause}$ function returns after the complete visit of a
hierarchy of groups. This visit is performed by one or more tasks
that run concurrently with all other tasks. Note also that multiple
notifications for a same group may be simultaneously running if a
group was paused or awakened multiply. Notifiers are not run in
mutual exclusion.
Let us note that the garbage collector should reclaim unreach-
able groups that are both exhausted and without subgroups. Indeed,
such groups cannot be awakened since they are unreachable, and no
task can be created under their sponsorship according to our model.

3.2 Threaded Implementation
We have implemented the single space model by modifying PPRC,
the Portable Posix Common Runtime [52], a user-level preempt-
tive thread package that comes with Boehm and Weiser’s garbage
collector [17]. We elected to modify this thread package as it was
already adopted for NeXeme [32], a distributed dialect of Scheme
based on the library Nexus [13].
A PPRC thread datastructure is given an extra field which points
at its group. By default, the value of this field is NIL, which means
that the current thread is not running under the sponsorship of a
group, and that it has the usual PPRC semantics. If not NIL, the
current thread is under sponsorship of a group. This approach
allows us to write applications where some threads may have the usual
PPRC semantics, while others follow Quantum semantics. Every
time a thread is scheduled, the energy of its associated group, if it
exists, is decremented. Therefore, in the current implementation, a
unit of energy is defined as a time slice of the PPRC scheduler.
The whole idea of Quantum is that the scheduler should pre-
empt a thread every time its associated group has no energy left.
However, like exceptions, preemption cannot be synchronous: if a
thread is in a critical section, preemption might lead to a deadlock.
Therefore, we used a similar technique as for asynchronous signals
in PPRC, where threads check for pending signals when exiting
critical sections. Two extra-fields per thread are used for this pur-
pose: a field indicating that Quantum preemptions are allowed, and
another field indicating that a Quantum event is pending. When a
thread exits a critical section, it checks if the Quantum event
flag was set by the scheduler and if preemptions by Quantum are
allowed.
According to the semantics, a new thread is created every time
Quantum notifies an event. In order to reduce the cost of thread
creation at notification time, we spawn a special thread whenever
we create a new group; this special thread is associated with the
group, and is nicknamed as the “group supervisor thread”. When a
user thread exits a critical section and detects a Quantum event, it
awakens the group supervisor thread associated with its sponsoring
group, which in turn runs the user specified specification. Care
is taken that only one thread per group can awaken the group su-
ervisor thread per Quantum event. A group supervisor thread is
maintained alive as long as its associated group is still in its useful
period. This organisation minimises changes in the PPRC kernel,
and Quantum preemption and resumption can be implemented as user-level code.

In addition, every primitive that accesses a critical section was modified so that Quantum preemption could not occur during their evaluation. In particular, this concerns I/O routines and the memory-allocation procedure GC_malloc [17].

This asynchronous notification slightly changes the semantics: indeed, energy may now become negative when the execution of a primitive overruns the energy left in its sponsoring group. We consider that it is not a real problem as the user is not given access to any construct that prevents Quantum preemptions; some primitives only, with a bounded length, are allowed to delay preemption.

The marginal cost of supporting Quantum is not high. Every time a thread is scheduled, the existence of a sponsoring group is tested, its associated energy is decremented, and a flag is set if it becomes negative.

### 3.3 Distributed Space Model

Distribution introduces multiple disjoint spaces of values linked by remote pointers and communicating by messages. We assume that distribution is introduced by a construct that creates a task on a remote site. Active message-passing style primitives [13] or remote function calls [30, 32, 42] are both compatible with Quantum. All sites have a scheduler managing local tasks.

The previously exposed implementation may be extended easily to cover distribution after introducing the notion of group brotherhood. A group stands for several tasks on several sites. To cope with distribution, a group is split into a hierarchical family of local groups ordered by a brotherhood relationship, each of them being local to a site. When a group g (see Figure 9) is created by call-with-group on Site1, it is known as the eldest local group of a family so far containing one brother only. When a task sponsored by g creates a remote task on Site2, a younger local group g1 is created on Site2 with g as elder brother. Symmetrically, g adds g1 as younger brother. When a subgroup g' is created by call-with-group on Site2 under the sponsorship of g1, its parent will be g.

In summary, a local group belongs to a site and holds the following information: (i) its elder brother, (ii) its direct younger brothers, (iii) its direct local tasks, i.e. the evolving set of tasks that run under its sponsorship on the local site, (iv) its direct local subgroups, i.e. its locally created subgroups.

A group is represented by at most one local group per site; local groups are implementation entities, but only eldest local groups are first-class values that incarnate the group they stand for. Therefore, in addition to the information of a local group, an eldest local group also knows: (i) its parent group, (ii) its exhaustion and termination notifiers.

The key point of the distributed implementation is that brethren groups interact as groups in parenthood relationship except that they use implementation-defined notifiers instead of user-defined ones. When a younger local group runs out of energy, it asks its elder brother for more energy. When a younger local group terminates, i.e. when it has no subgroups, no younger brothers, and no tasks, it posts a notification (accompanied with its remaining energy) to its elder brother which then removes it from its younger brothers. When an eldest local group terminates, it reacts as previously described with respect to its parent group. The difficult case is when an eldest local group runs out of energy, since it should not post a notification to its parent unless all its younger brothers are also out of energy. The solution is then to dry the tanks of the younger brothers (but not the subgroups) to the benefit of the eldest brother. A group has no centralised tank but a collection of local tanks connected by implementation-defined notifiers. These inner transfers of energy are invisible to the user; they may use sophisticated techniques to ensure load balancing of energy inside the connected tanks of a group.

The same schema may be used to provide each task with its own tank, hereby

---

---

---

---
local group is allowed to notify its parent of its exhaustion.

3.4 Distributed Implementation

A prototype distributed implementation has been implemented using the library Nexus [13]. Nexus is structured in terms of the following basic abstractions: sites\(^4\), threads, global pointers, and remote service requests. A computation executes on a set of sites and consists of a set of threads, each executing in an address space. An individual thread executes a sequential program, which may read and write data shared with other threads executing in the same site.

The global pointer (GP) provides a global name space for objects, while the remote service request (RSR) is used to initiate communication and invoke remote computation. A GP represents a communication endpoint: that is, it specifies a destination to which a communication operation can be directed by an RSR. GPs can be created dynamically; once created, a GP can be communicated between sites by including it in an RSR. A GP can be thought of as a capability granting rights to operate on the associated endpoint. Issuing an RSR causes the data buffer to be transferred to the site designated by the global pointer, after which the routine specified by the handler is executed, potentially in a new thread of control. Both the data buffer and pointed specific data are available to the RSR handler. Nexus was ported to the thread package PPCR in order to implement NeXeme, a dialect of Scheme offering a functional version of GPs [32].

As described in Section 3.3, groups are incarnated by local groups on each site; GPs pointing at local groups can be passed between sites. Operations like pause and awake are allowed on remote groups using RSRe. Nexus primitives that use mutual exclusion, like buffer management, were defined so as to prevent preemption by Quantum; also, the thread that polls for incoming messages in Nexus does not run under the sponsorship of a group in order to prevent deadlocks.

We have implemented a weaker version of the semantics for the distributed version. Local groups follow a hierarchical organisation according to a brotherhood relationship. In the current implementation, an exhaustion event is notified whenever a local group has no energy left without taking into account the energy available in its younger brothers. For instance in Figure 9, when there is no energy left in g1 on Site2, a notification is run on Site1 under the sponsorship of g1 trying to transfer more energy to g1. If not enough energy is available in g, a user-specified notification is run in the parent of g. These notifications proceed even though there might be some energy left in the group g3 on Site3. A future implementation will provide a transparently distributed notion of group on top of the current system.

3.5 Performance Evaluation

In order to evaluate the performance of the implementation, we wrote several programs whose purpose is to identify the cost of Quantum operations. The code run in our experiment is available from the URL http://www.ecs.soton.ac.uk/~lavm/Quantum. We distinguish the shared-memory multi-threaded implementation from the distributed implementation.

The first experiment measures the cost of exhaustion notification and group awakening in a shared memory. The program SM-notif creates a group that sponsors a task executing an infinite loop. The group is initially given one unit of energy; its exhaustion handler awakens the group by transferring one unit of energy and also updates a counter indicating the number of exhaustion notifications.

In the current implementation, the infinite thread of SM-notif is suspended as no energy is left in its sponsor, its group supervisor thread is called, which in turn resumes the thread. We compare this operation with an operation for which we intuitively know the cost. In SM-ping-pong, two threads access a critical section in turn.

Figure 10 shows the average cost of one operation over 1000 executions. Experiments were run on Indy workstations, Mips R4600, running IRIX 5.3, using 10Mb ethernet. We give the user and system times per operation, and their sum is used for comparison purpose. We see that notification detection and awakening of a group is approximately twice the cost of two threads accessing a critical section in turn.

Remark We use a modified version of the user-level thread package PPCR. Therefore, the time spent is the thread package for scheduling, mutual exclusion, etc. appears in the user time. Also the total time includes the time spent by the Nexus library to poll for incoming messages.

\[
\begin{array}{cccccc}
\text{Program} & U (\mu s) & S (\mu s) & U+S (\mu s) & \text{ratio} \\
\text{SM-ping-pong} & 0.79 & 0.13 & 0.92 & 1.00 \\
\text{SM-notif} & 1.88 & 0.31 & 2.19 & 2.38 \\
\text{DM-ping-pong} & 11.62 & 3.09 & 14.71 & 1.00 \\
\text{DM-notif} & 16.43 & 2.96 & 19.39 & 1.31 \\
\text{SM-pause-awake} & 12.90 & 0.80 & 12.80 & 1.00 \\
\text{DM-pause-awake} & 80.02 & 15.60 & 95.62 & 7.47 \\
\end{array}
\]

Figure 10: Performance

The next experiment evaluates the cost of notifications when groups are distributed. Two sites \(S_1\) and \(S_2\) are used in the program DM-notif. An initial group is created on \(S_1\), and a young brother group is created on \(S_2\) by a remote service request; the task sponsored by the young brother immediately consumes all its energy. A notification transfers more energy from \(S_1\) to \(S_2\). Again, we compare the cost of exhaustion detection and awakening of distributed groups with an operation for which we intuitively know the cost. The program DM-ping-pong measure the cost of sending a remote service request to \(S_2\) and of receiving an acknowledgement in the form of a remote service request.

Again, Figure 10 shows the average cost of one operation over 1000 executions. We see that the combined cost of exhaustion detection and energy transfer is only 30% higher than the cost of a communication and its acknowledgement. Also, the distributed memory operation is roughly an order of magnitude more expensive that the shared memory one.

In order to measure the performance, we turned off the sending of messages for the distributed garbage collector, but we maintained garbage collection on each site.

The third experiment evaluates the cost of pausing and awakening groups in a shared memory system. The program SM-pause-awake creates a group \(g\) which has two subgroups \(g_1\) and \(g_2\). Each subgroup sponsors a threads that is an infinite loop. The program repeatedly pauses the group \(g\) and its subgroups, and awakens them with the same distribution of energy. The results for this experiment varied extremely (as opposed to the others which were rather stable). The reason is that the program does not check that a group hierarchy has been completely awakened before being paused again. It is also

\(^4\)Nexus distinguishes nodes from contexts; for the purpose of this paper, it suffices to consider that sites are processes.
important to note that the pause primitive creates a new notification thread for each subgroup (as opposed to other notifications which are run by the group supervisor thread created at the same time as the group). This program has therefore a serious effect on memory allocation, and the results displayed are an average over 100 executions.

The distributed variant is DM-pause-awake which involves two sites $s_1$ and $s_2$. The group $g$ is created on site $s_1$ and initiates a remote service request to $s_2$ which results in a younger brother, sponsoring two subgroup $g_1$ and $g_2$. Similarly, $g$ and remote subgroups are paused and awaken. We observed that the cost is only 8 times the cost of the shared memory implementation.

### 4 Discussion and Related Work

Our semantics measure the resources used by computations according to an independent cost model. One might prefer to have other cost models, taking into account, for instance, memory occupancy (duration or size). In order to have a finer control on energy consumption, we are extending Quantum so as to let the user create and manage different budgets from which energy can be consumed; e.g. geographical extent, computation resource, ... Therefore, we can regard energy as any multidimensional (vector) data structure of budgetised resources. Another interesting question is to decide how garbage collection should be billed in a multi-user environment. At the moment, garbage collection is part of the administration cost of the system and is indirectly billed via the cost of allocators.

To the best of our knowledge, three different communities have studied techniques to control computations: the programming languages, the parallel and distributed systems, and the agents communities.

#### 4.1 Engines and Sponsors

Our notion of group is at the intersection of two different ideas: Haynes and Friedman’s engines and Kornfeld, Hewitt, and Osborne’s sponsors, which we develop below.

Haynes and Friedman [19, 20] introduce the engine facility to model timed preemption; variants can also be found in [8, 9, 48]. Engines differ from our groups in a number of ways. Engines are defined in a sequential framework and are used to simulate multiprogramming. Since engines do not deal with parallelism, they do not offer control facilities like pause and awake. Another major difference is that a given engine can be executed several times, while a group can only be executed once. Using continuation terminology, engines are “multishot”, while groups are “single-shot” [4]. A group is a name and an energy tank for a computation, but, unlike an engine, it does not embody its continuation. Our decision to design “single-shot” groups is motivated as follows. The ability to restart a same computation several times is an unrealistic feature for a distributed language because the computation may be composed of several tasks distributed over the net. Haynes and Friedman also propose nested engines, i.e. engines that can create other engines. In their approach, nested engines have the same temporal vision of the world, because each computation consumes ticks, i.e. energy quanta, from all parent engines (direct and indirect). On the contrary, groups offer a more distributed vision of the world, because groups are tanks, from which local tasks consume energy.

Kornfeld and Hewitt’s sponsors [20], Osborne’s enhanced version of them [36, 37, 18], and subsequently Queinnec’s groups [45, 43], also allow the programmer to control hierarchies of computations in a parallel setting. Osborne’s sponsors are entities that give attributes, such as priority, to tasks, which can inherit attributes from several sponsors. A combining rule yields the effective attributes of a task, and then determines the resources allocated to the task. If the group hierarchy changes, priorities should be recomputed, which can be costly, especially in a distributed environment. With Quantum groups, scheduling of a task is only decided by examining the energy available in its only sponsoring group, which is local. Furthermore, priority is a difficult notion to grasp in a heterogeneous environment, while total work accomplished is more intuitive. In particular, our cost model allows us to program applications searching the best solution at a given amount of energy, i.e. at a given cost. Queinnec’s Icsla language has a notion of group which substantially differs from the one presented here. As Icsla is energy-less, pausing a group does not collect energy and can be performed lazily. Also, Icsla does not have any of the notifications of Quantum. Let us observe that termination notification is a generalisation of unwind-protect [49]. Hieb and Dybvig [21] spawn operator returns a controller, which can be invoked to suspend or restart part of a computation tree; their approach relies on a notion of partial continuation.

#### 4.2 Parallel and Distributed Computing

Unix-like operating systems offer a different model to control processes: a process is given a unique identifier which can be used to send signals to it, e.g. kill. We believe that this model of control does not provide the appropriate abstraction [37]. Indeed, we want to be able to control computation, but we might not know the processes it is composed of, because the code we execute was not written by us. By associating groups with computations, we abstract from the details of execution. In addition, our model is designed for running in a distributed framework, so that groups can control tasks over different machines.

PVM [14] extends the Unix model to distribution. Each task enrolled in PVM has a unique identifier. PVM has a primitive for sending signals to remote tasks. PVM has also a notion of group, which tasks can dynamically join or leave. Groups are used in PVM for broadcasting messages or to perform synchronisations. MPI-2 [15], the new version of MPI [12], also offers the possibility to send signals to groups of tasks. Some resource management tools [41] deal with memory allocation, CPU allocation, and load balancing. However, we believe that there is no equivalent to the Quantum facilities in those message-passing libraries, like the ability to resume and suspend computations, or dynamic transfer of resources. This does not mean that such a semantics cannot be programmed on top of message-passing libraries; signalling and broadcasting could be used to implement Quantum.

The Propero Resource Manager [35] comes as a library to allocate tasks to processors in a multi-processor architecture. In Quantum, we could regard locations, i.e. processors, as a resource. Quantum would then offer a high-level interface to Prospero, which would take care of practically assigning processors.

#### 4.3 Agents

The mobile agent community deals with the problem of controlling distributed mobile computations, called agents. The most widespread agent systems are Telescript [27, 28], Tacoma [23, 24], Agent-TCL [16], and Ara agents [40, 39]. Very few of them are able to control resources in a similar way as Quantum.

Telescript [27, 28] is an agent scripting language. Every Telescript agent has a permit that limits its capabilities in the electronic marketplace, like for instance, the right to use a resource, or a maximum overall expenditure of resources measured in teleticks.
Quoting [28], "if the agent exceeds any of its quantitative limits, the engine destroys the agent unceremoniously. No grace period is extended". Our model is more general than the Telescript approach as it allows us to drive computations using \texttt{pause} and \texttt{awake} and to monitor them using asynchronous notifications. Also, our model supports agents that perform parallel and distributed computations, what is usually called multi-agent systems.

Ara agents [40, 39] are equipped with resource accounts called \textit{allowances}, which are similar to our groups [40, p. 6]; as indicated by Peine, they are a recent concept, not complete yet. Several agents can share the same allowance, also called a group. Agents can transfer resources explicitly between accounts. In Ara, agents can be suspended or reactivated; these actions however are performed on agents directly and not on groups [40, p. 23]; as a result hierarchical computation cannot be controlled as in Quantum. In Ara migration operations specify the allowance that an agent should take with it [40, p. 29]. We adopt a different view as our groups are transparently distributed. Ara agents are not notified of an exhaustion [40, p. 35]; Peine consider that this is not a severe problem as an agent may enquire about the existence of a resource. We believe that this argument is not valid in parallel/distributed computing because another parallel computation might consume the resource.

Arthursson et al. [1] describe a mobile agent architecture based on the distributed language Erlang. They provide some form of resource control, but an agent is terminated once it has consumed all its resources.

In Agent TCL, an access list specifies the quantity of a resource that an agent can consume; but apparently, no mechanism is provided to transfer new resources to an agent.

Cardelli and Davies [5] present a set of combinators to program the WWW. Some of them deal with real-time operations, such as transfer of data on a medium that can fail. We are studying how Quantum could be extended to support such real-time operations.

5 Conclusion

In this paper, we present the paradigm Quantum, whose purpose is to monitor and control computations in a parallel and distributed framework. Quantum provides a model to control resources in distributed computations, and therefore can be used in agent scripting languages or to program resource administration applications. Besides, it offers the end user the possibility to control hierarchies of computations, to perform some load-balancing, and to introduce some resource-based priority between parallel computations; also, it provides primitives to build "any-resource" algorithms, applying the idea "any-time" algorithms [7] to any form of resource.

As an application, we are building an information-discovery agent [6], whose resources can be controlled and monitored using Quantum primitives. The agent explores HTML pages and is able to move across WWW sites, when following links. The interface to the WWW is provided by a special communication module of Nexus over HTTP. Quantum primitives allow us to delimit the geographic expansion of the agent, following the Time to Live (TTL) approach adopted in network protocols; it is also possible to suspend and resume the agent.

Quantum is sound because it prevents generating energy. Furthermore, it has some features that can be used to enforce security in the application: (i) energy cannot be generated, but can only be transferred between computations; all "accounting" operations remain under absolute control of the primitives; (ii) groups are the only handle to control computations, and scoping rules of the adopted language guarantee that groups will be visible only where the programmer wishes them to be; (iii) there is no primitive that returns the group in which the user code is running, which ensures that user code cannot control its sponsoring group, and hence it cannot control tasks running in parallel with it, unless explicitly passed handles to their sponsoring groups. Security is an important issue in distributed agent-style applications. Using Quantum primitives, there is nothing that prevents users from erroneously transferring resources between groups, or making a group accessible to and then preemptable by another task. However, we believe that some static analysis [50] would be able to detect whether a group might become preempted if made accessible in a public data structure for instance.

Quantum is the core of a consumption-oriented language which is particularly suitable to programming over the Internet. In the future, we plan to investigate a fault-tolerant version of the language, which would be energy aware.

6 Acknowledgements

Thanks to Jonathan Dale for his useful comments. Andreas Kind and Jean-Pierre Briot mentioned a link between Quantum and anytime functions. Luc Moreau was partially supported by EPSRC GR/K30773 and EC project reference ERB 4050 PL 930186, and Christian Quinmec was partially supported by GDR-PRC de Programmation du CNRS and EC project reference ERB 4050 PL 930186.

Bibliography


