Distributed and Multi-Type Resource Management[*]

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

Christian Queinnec
Christian.Queinnec@lip6.fr
LIP6
4, place Jussieu, 75252
Paris Cedex
France

May 17, 2002

Abstract

The authors have previously defined Quantum, a framework for managing resources, which can be used by both providers and consumers of resources to express and program resource-related operations. In this paper, we extend this framework to the distributed setting by introducing distribution-specific operations, and we generalise the framework to support multiple types of resources.

1 Introduction

Dynamic code loading has popularised the idea of Internet servers able to reconfigure themselves and to extend their capabilities by uploading code dynamically — examples of such systems can be found in the mobile agent literature. The full power of this paradigm shift can be achieved if untrusted code can be run in a safe manner, and in particular if malicious code can be prevented from using too many resources. This raises the problem of resource management, both for the provider and the consumer of resources.

Another important trend is illustrated by multi-agent systems or services-based architecture (in particular in the Grid context), where complex applications are the result of dynamic composition, opportunistic reuse, and on-the-fly creation of multiple

[*]ECOOP'02 Workshop on Resource Management for Safe Languages, Malaga, Spain, June 2002.

[†]This research is funded in part by QinetiQ and EPSRC Magnitude project (reference GR/N35816).

distributed computations. Resource management is not only crucial, as illustrated by proposals for computational economies, but has now become a distributed problem.

In previous work [9, 10], the authors introduced *Quantum*, a framework generalising Kornfeld and Hewitt's group hierarchy [7] and providing a programmatic interface for managing resources in a distributed setting. Quantum is based on the notion of *energy*, an abstract notion denoting a quantity of resources, and on *groups* acting as tanks of energy. Groups are organised along a hierarchical structure. Groups *sponsor* computations, which consume energy from the group they are directly sponsored by. Two forms of notification are supported: *exhaustion* of the energy contained in a group and *termination* of the computation sponsored by a group. Additionally, Quantum provides a mechanism for pausing and resuming a hierarchy of computations. Notifications are made available to the programmer and therefore can be arbitrary computations, whose resources must also be managed: Quantum specifies how such notifications can be integrated in a single framework. Our previous work focused on its formalisation [9] and its implementation in a shared memory [10].

The distributed aspect of Quantum had not been investigated properly. We assumed that the shared memory could be extended to the distributed context, but this required complex and unpractical algorithms to maintain tank levels transparently. Since then, advances in mobile agent systems, and "Internet Programming" languages have shown that distribution has to be represented explicitly in the formalism.

While our model of resource management had always been intended to support multiple resources of different types (and not just processor resources), we had never shown how Quantum could be extended to accommodate different types of resources.

The contribution of this paper is twofold: *(i)* the introduction of two different primitives related to distribution — migration and communications — and their semantics in terms of groups. *(ii)* the support for multiple types of resources. Each of these contributions is presented in turn, followed by a brief related work section and a conclusion.

2 An Overview of Quantum

In this section, we overview Quantum as described in previous publications [10, 9]. 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 threads, or higher-level constructs such as **pcall** or **future**. In the sequel, we shall used the term *thread* to denote an evaluation thread created by the constructs for parallelism.

Our goal is to be able to allocate resources to computations, and to monitor and to control their use as evaluations proceed. We regard two events as essential to the lifetime of a computation, which may trigger customisable actions. The *termination* of a computation marks the end of its life, and we would expect unconsumed resources to be transferred to a more suitable computation. The *exhaustion* of the resources allo-

$$
\begin{array}{rcl}
\textit{primitives} & ::= & \mathsf{call\text{-}with\text{-}group}(F, e, \varphi_e, \varphi_t) \\
 & & \mid\ \mathsf{pause}(g, \varphi_p) \\
 & & \mid\ \mathsf{awake}(g, e) \\
g & \in & \textit{Group} \\
e & \in & \textit{Energy} \\
F & : & \textit{Group} \times \textit{Energy} \to \alpha \\
\varphi_e, \varphi_t, \varphi_p & : & \textit{Group} \times \textit{Energy} \to \textit{void}
\end{array}
$$

Figure 1: Abstract Syntax of Quantum Primitives

cated to a computation may trigger 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 introduce 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 threads 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* [7, 11, 6] the computation it is associated with. Reciprocally, every computation has a sponsoring group, and so does every thread.

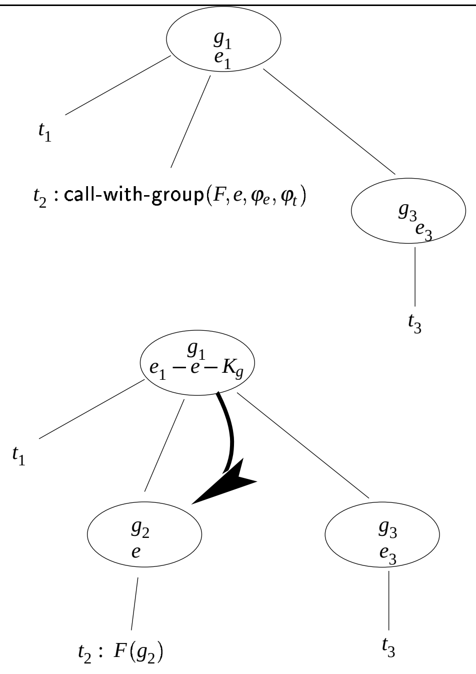


Figure 2: Group Creation

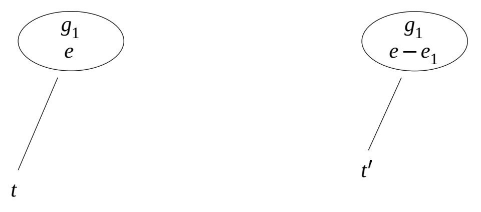


Figure 3: Computation and Energy Consumption

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

$$\mathsf{call\text{-}with\text{-}group}(F, e, \varphi_e, \varphi_t)$$

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 threads t_1 and t_2 and a subgroup g_3 itself sponsoring a thread 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 $e_1 > e + K_g$.

> **Remark** In Quantum, every action has an associated cost. Figure 2 shows that the energy of g_1 is $e_1 - e - K_g$ after transition. The value $-e$ is the amount of energy transferred to the new group g_2 and $-K_g$ represents the cost of the group creation operation. $\Box$

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 thread t 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*) φ_t and φ_e, respectively[1]. A group is said to be terminated, when it has no subgroup and it does not sponsor any thread; i.e. no more activity can be performed in the group. In Figure 4, when the only thread t_4 of group g_3 is terminating, the function φ_t is asynchronously called with g_3 as argument to notify its termination, and the energy surplus of g_3 is transferred back to g_1. Note that the execution of the notifier φ_t is sponsored by g_1, i.e. the parent of g_3.

[1]Subscript t denotes termination, whereas subscript e denotes exhaustion.

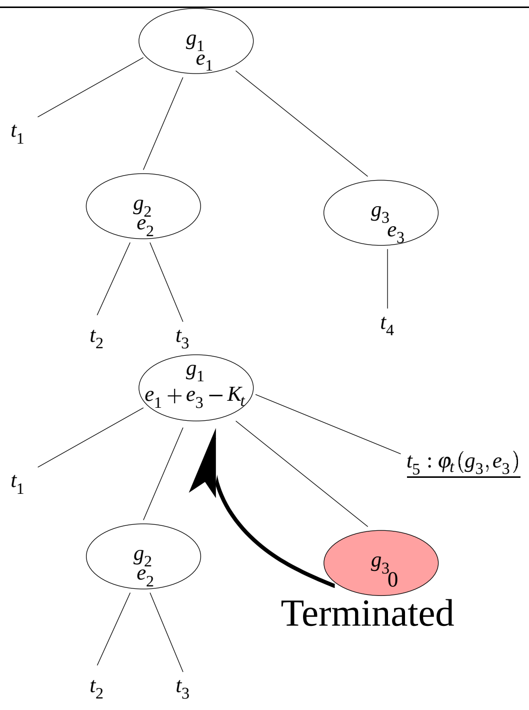


Figure 4: Termination of a Group

In Figure 5, a computation t_2 sponsored by g_2 requires more energy than available in g_2; the function φ_e 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 $\mathscr{Q}$uantum 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 threads that it sponsors can proceed as long as they do not require more energy than available. Asynchronous notifications are represented by dotted lines. Once a computation requires more energy than available in its sponsoring group, the state of its group changes to exhausted, and at the same time an asynchronous notification φ_e is run. When all the subgroups and all the threads sponsored by a group terminate, its state becomes terminated, while the asynchronous notifier φ_t is called. Let us observe that the terminated state is a dead end in the state diagram; this guarantees the *stability* of the termination property: once a computation terminates, it is not allowed to restart (as the resource that it did not consume may have been reallocated).

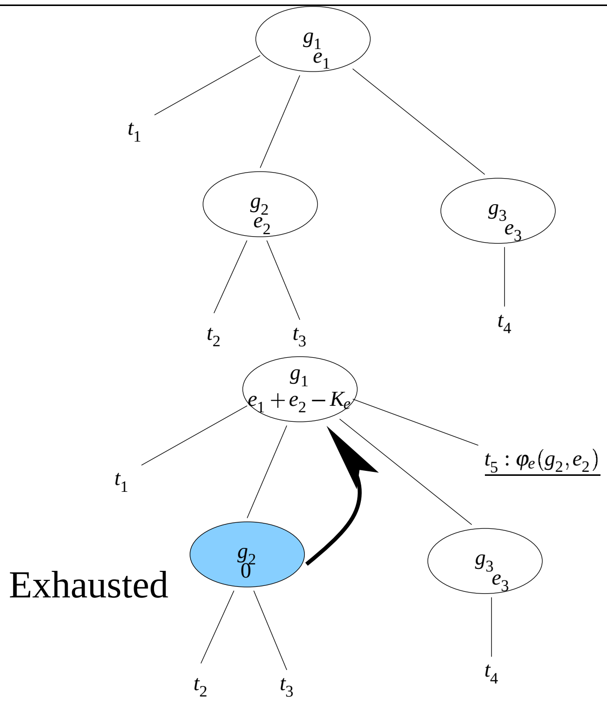


Figure 5: Exhaustion of a Group

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 $e_1 > e + K_a$ and g_2 is not terminated.

Let us observe the asymmetric behaviour of pause and awake: the former operates recursively on a group hierarchy, while the latter acts on a group and not its descendants. However, we might wish to awake a hierarchy recursively, for instance when we wish to resume a paused parallel search. In particular, we might wish to resume the search with the energy distribution that existed when the hierarchy was paused. Unfortunately, such information is no longer available because groups are *memoryless*. By this, we mean that a group does not remember the amount of energy it had before being paused. It is therefore the programmer's responsibility to leave some information at pausing-time about the way a hierarchy should be awakened. Not only does pause transfer energy, but it also posts a notification for each group in the tree. Figure 8 displays the precise behaviour of pause. Evaluating pause(g_1, φ_p) forces into

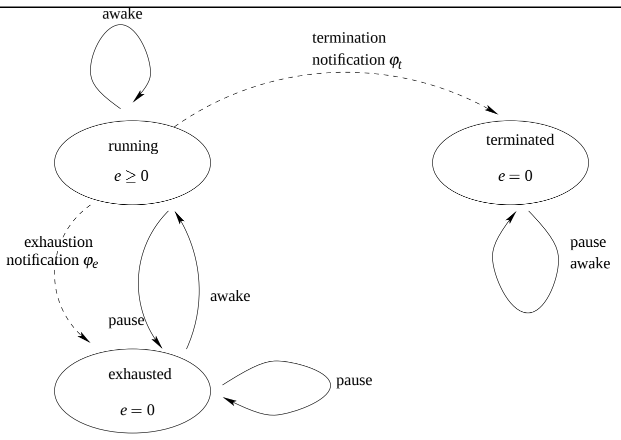


Figure 6: State Transitions

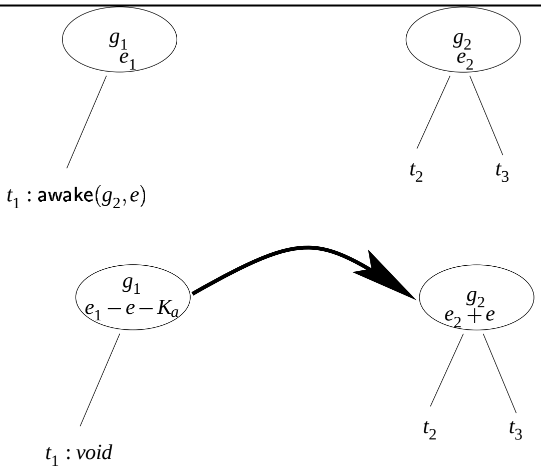
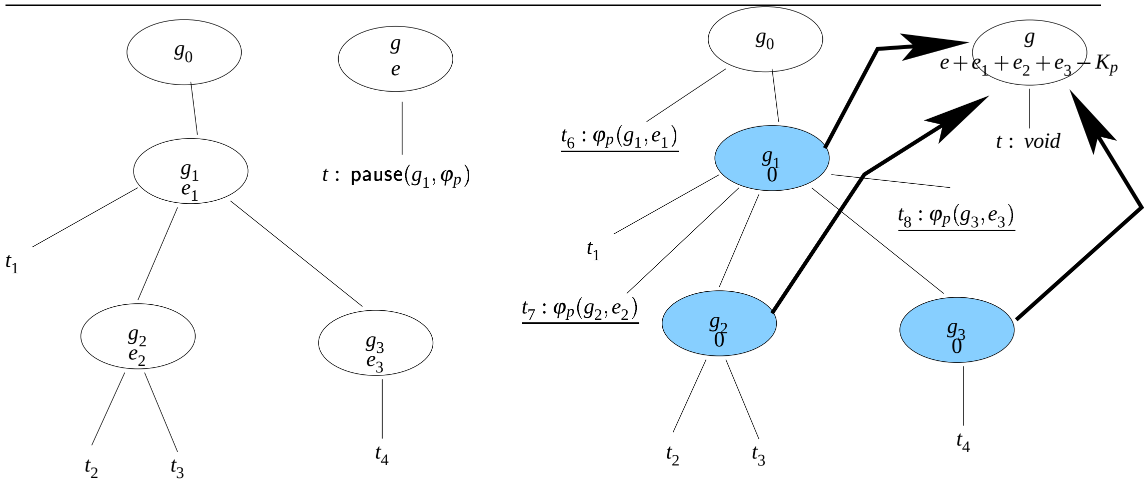


Figure 7: Awaking a Group

the exhausted state each group g' in the hierarchy rooted by g_1; moreover, for each g', an evaluation that calls φ_p with g' as argument is created under the sponsorship of the parent of g'. Let us note that notifications are prevented to run as all groups in the hierarchy have been dried out (except the notification on the root g_1, which is sponsored by g_0, the parent of g_1 and then might run). Once the root of the hierarchy is awakened, any notification sponsored by the root will be activated, and may decide to awake the group it is applied on, and step by step, energy may be redistributed among the hierarchy.

This section concludes our overview of Quantum. We now investigate how it can be extended to the distributed setting and how we can support multiple resources.



g_0
g e
g_1 e_1
$t : \mathsf{pause}(g_1, \varphi_p)$
t_1
g_2 e_2
g_3 e_3
t_2
t_3
t_4
g_0
g $e + e_1 + e_2 + e_3 - K_p$
$t_6 : \varphi_p(g_1, e_1)$
$t : void$
g_1 0
$t_8 : \varphi_p(g_3, e_3)$
t_1
$t_7 : \varphi_p(g_2, e_2)$
g_2 0
g_3 0
t_2
t_3
t_4

Figure 8: Pausing a Group

3 Distributed Resource Management

As far as distribution is concerned, we distinguish the transfer of data between hosts from the transfer of groups between hosts. The former can easily be expressed by send and receive primitives ‘à la’ π-calculus. The latter is reminiscent of remote procedure calls and migration of mobile agents. Indeed, groups act as sponsors of computations; if a group changes location, so does the sponsored computation.

In this work, we do not want to impose strong mobility to a programming language for the sake of resource management (nor do we require the power of first-class continuations). We introduce a primitive for migrating a group that is similar to the invocation of a remote procedure.

We introduce the primitive $migrate(h, f)$, which requires two arguments: h a host name and f a procedure without argument (a thunk). The effect of the *migrate* primitive is displayed in Figure 9.

Before transition, on host h_1, *migrate* is called with arguments h_2 and f, with a current stack noted as an evaluation context E_1 [4], and with a sponsoring group g_1 containing e_1 units of energy. After transition, the group g_1 contains no energy and sponsors no computation; it can be seen as a “zombie” [12] which acts as a handle to the remote computation. On host h_2, a new group g_2 is created to sponsor the application of the thunk f; group g_2 contains e_1 units of energy, minus the amount K_m necessary for the migration operation. Let us note that after transition, the execution thread on h_1 has disappeared, and on h_2 we have *not* reactivated the execution context E_1: this is in effect a weak migration. For migration to proceed, we require *migrate* to be executed in a group that sponsors *only one* thread.

The parent of group g_2 is g_1. In Figure 9, we represent g_2 as a dashed ellipse, because this group is not explicitly created by the user using the group-creation primitive

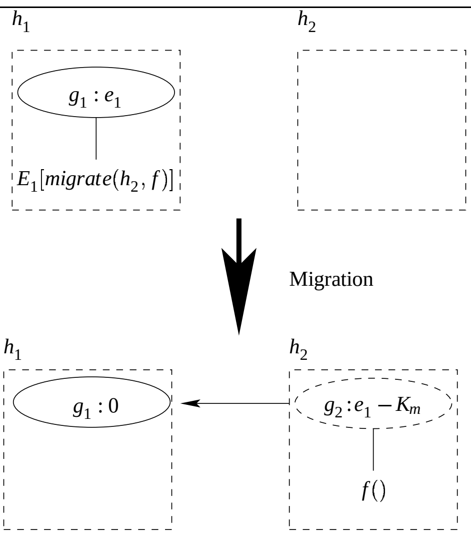


Figure 9: Migration

call-with-group, but it results from a migration. Such a group is referred to as a *remote group*. For a remote group, the handlers for termination and exhaustion are defined by the semantics so as to provide the behaviour described in Figure 10 (a) and (b).

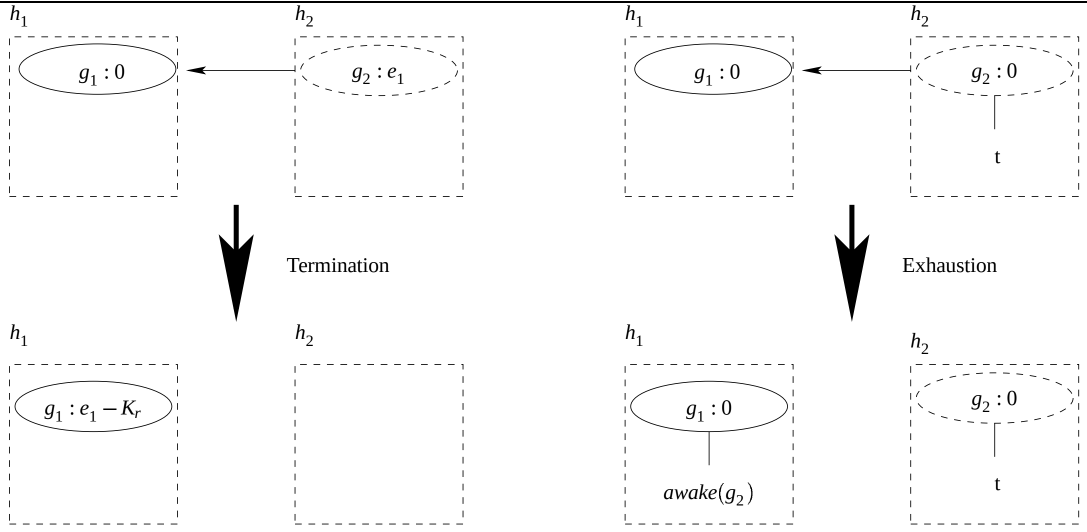


Figure 10: Return from Migration: (a) Termination — (b) Exhaustion

In Figure 10 (a), when a remote group g_2 detects the termination of the computation it sponsors, its energy is transferred back to its parent g_1, minus the amount *kkostreturn* necessary for the return. After such a transition, termination can then be detected in g_1.

According to Figure 10 (b), exhaustion of g_2 triggers a propagation of an exhaustion notification to host h_1, under the sponsorship of g_1. As such a computation attempts to run under g_1, which contains no energy, a notification will also be raised, to be run under the sponsorship of g_1's parent. If more energy is transferred to g_1, it will then be transferred to g_2, through the use of the *awake* primitive.

The reader may observe that, in Figure 10 (b), the activation of a notification under the sponsorship of g_1 takes place without being accounted for. We note that a group exhaustion occurs only once (before the group gets refilled again), and therefore the cost of notification propagation can be included, i.e. "precharged", in the cost of group migration K_m.

Nothing prevents the thunk activated by a migration to call the *migrate* primitive again. Such successive migrations create a sequence of nested remote groups, reminiscent of forwarding pointers left by mobile agents. There is an opportunity to shortcut such sequences of remote groups using a mechanism similar to the collapsing of chains of pointers [8].

The rationale for requiring a single thread in a group g_1 before allowing migration to proceed is the following. After migration, g_1 acts as a handle for the remote group g_2: exhaustion of g_1 implies that g_2 is exhausted; if more energy is transferred to g_1, it is passed on to g_2. Additionally, chains of such handles can be shortcut easily.

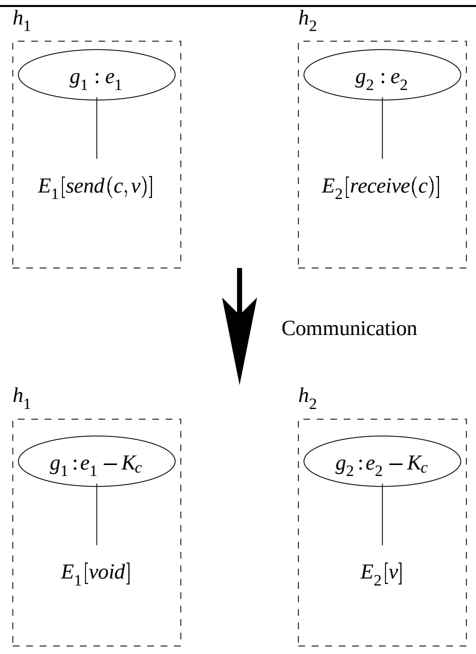


Figure 11: Synchronous Communications

Figure 11 displays the semantics of synchronous communications using primitives *send* and *receive*. No group is created or migrated here; instead, both sponsoring groups are charged with the cost of communication K_c.

4 Management of Multiple Resources

Section 3 and our previous work focused on a single type of resource, namely processor time. In particular, in [9], we introduced an abstract notion of *energy* to denote a quantity of resources; we defined its meaning as processor time using a notion of "tick" reminiscent of the one used in Scheme engines [3]. In this Section, we step backward and study the primitives manipulating energy. We then show that they may be extended to cope with various (and multiple as well) types of energies. Some design considerations are also addressed in this Section.

4.1 Operations on energy

From an energy system implementor's viewpoint, there are only three primitive operations that deal with energy tanks. These operations target how energy is *(i)* merged when a subgroup terminates and gives its energy back to its parent, *(ii)* consumed while a group performs some work, and *(iii)* split between the creating and created groups.

The following Java interface represents these operations. The receiver of these messages, an energy-tank, is side-effected by these operations.

```
interface EnergyTank {
  void    merge (Energy energy);
  boolean consume (Energy delta);
  Energy  split (SplitDescriptor d) throws EnergyExhaustion;
}
```

The simplest operation is *merge* that occurs when a group terminates and returns its remaining energy to its parent. The two energies are then recombined and put back in the parent's tank (the receiver of the *merge* method). A little less simpler is the *consume* operation that is repeatedly called to decrement the energy tank of the working group. The argument is a (request for) energy and the answer is a boolean telling whether the request is accepted. When the request is accepted, the energy tank is decremented, that is, consumption is pre-paid. To support concurrent behaviour, the request and the decrementation are performed atomically.

Creating a group is a more complex operation since the energy of the current group has to be split into two parts under the user's control. The energy returned by *split* will fill the subgroup's tank, this returned energy is removed from the energy tank of the working group (the receiver of the *split* method). The *split* operation takes an argument, a descriptor, specifying how the user wants to split energy. The descriptor is any type of data that users may safely create to carry their intention (a string, a percentage, etc.). It cannot be an energy since, for safety reasons, energy should never be handled directly by users since users may duplicate it, lose it, steal it, etc.

The *split* operation interprets the descriptor and returns some energy to fill the freshly created group with the appropriate part of the current energy. In terms of energy, if we merge back the split energy into the current tank, we should not create energy ex nihilo. Thus the following invariant, *informally* stated, holds:

$$\texttt{e.merge(e.split(d))} \leq \texttt{e}$$

While there is an obvious mapping from energy to numbers specially when energy is CPU-time or printed pages, other kinds of energy may be expressed using the previous model. Let us give two different examples illustrating how differently resources can be merged, consumed and split.

File Permissions The right to read a specific file may be represented as a boolean energy. We will consider that the *consume* operation does not exhaust this right[2]. The *merge* disjuncts the two booleans. The *split* operation receives one of three possible descriptor strings: “keep”, “give” or “share”. These descriptors allow the new group’s creator to keep, give or share their right. Table 1 presents an algebraic view of the operators. Without the permission to read a file, the split operation always return (f, f) for any split descriptor (denoted by $*$ in the last line of Table 1).

merge	$E_1 \times E_2 \to E_1 \vee E_2$
consume	$E_1 \times \Delta E_2 \to E_1$
split	$t \times \text{keep} \to (t, f)$
	$t \times \text{give} \to (f, t)$
	$t \times \text{share} \to (t, t)$
	$f \times * \to (f, f)$

Table 1: The boolean energy to read a file.

Migration Hops We may count the number of hosts a computation is allowed to migrate to in order to restrict the diameter of a distributed computation. The *consume* operation triggered by migration decrements the number of allowed hops. The *merge* operation ignores the number of hops any terminating subgroup may return to it and sticks to the initial number of hops it received when created. The *split* may restrict the number of hops it gives to the new group while preserving its own number of allowed hops. The restriction may be expressed with a natural number (p in Table 2). These definitions allow diameters of distributed computations to be properly nested.

4.2 Multiple energies

Though hidden from users, energy should be managed by implementors. The implementation of the interface `EnergyTank` relies on some operations on `Energy` so it

[2]Alternatively, the write permission on a write-once device would consume the energy at the first successful attempt to write.

merge	$E_1 \times E_2 \rightarrow E_1$
consume	$E_1 \times \Delta E_2 \rightarrow E_1 - 1$
split	$n \times p \rightarrow (n, min(0, n-p))$

Table 2: The energy for hops.

is possible to add, subtract or split energies. Although Object-Oriented, the next interface has no side-effect (as was the case with a tank of energy), it deals with pure energy.

In the next interface, the *quotient*, *plus* and *minus* operations on energy support the *split*, *merge* and *consume* tank operations.

```
interface Energy {
  Energy quotient (SplitDescriptor d);
  Energy plus (Energy e);
  Energy minus (Energy e) throws EnergyExhaustion;
}
```

We may now implement a simple `EnergyTank` as:

```
class EnergyTankImpl implements EnergyTank {
  private Energy _e;
  EnergyTankImpl (Energy e) {
    this._e = e;
  }
  Energy split (SplitDescriptor d) throws EnergyExhaustion {
    Energy gift = this._e.quotient(d);
    this._e = this._e.minus(gift);
    return gift;
  }
  void merge (Energy e) {
     this._e = this._e.plus(e);
  }



  boolean consume (Energy de) {
     try {
       this._e = this._e.minus(de);
       return true;
     } catch (EnergyExhaustion ee) {
       return false;
    }
  }
}
```

The `EnergyTank` interface copes with one type of energy only. It is a simple step to extend such an interface to multiple energies, say `EnergiesTank`. First, we suppose an `EnergiesTank` to hold a set of `EnergyTank`s. Second, we transfer sets of energies. Third, the *split* operation now takes a set of split descriptors and it is possible from any of them to determine which type of energy it splits.

```
interface EnergiesTank {
  void     merge (Energies energies);
  boolean  consume (Energies delta);
  Energies split (SplitDescriptors d) throws EnergyExhaustion;
}
```

The final step is to let the user devise new types of energy (and associated split descriptors) and have the underlying machinery manage it. It remains to let the user create an initial amount of the new energy and register it with the current group. Once registered, this new energy will be managed entirely by the group according to the user's split descriptors.

5 Related Work

A number of existing systems support resources accounting. Telescript [15] featured "clicks" that are a unit of charge deducted from an agent's account. JRes and JKernel [2] support accounting of memory, CPU and network usage. Nomads [13], through a modified JVM, supports strong migration of agents and resource accounting; in particular, a limit file is able to specify both quantity limits (such as disk space or memory) but also rate limits (such as disk usage rate and transfer rate). Java Seal2 [14] is an extension of Java Seal [1] which provides portable resource accounting. A notion of process is introduced in KaffeOS [5], a modified JVM, which allows resource control in a fine manner.

All these systems are complementary to Quantum: indeed, they implement the accounting of resource usage and they raise a notification when a resource quota is reached, while Quantum provides the mechanism to transfer (i.e. add or remove) resource dynamically between distributed computations. Quantum also provides a programming model to support the execution of asynchronous notifications, under the control of the same resource management system. Notifications are therefore becoming a key programming technique that can be made available to the programmer. Additionally, our model also supports resources they do not necessarily have a physical reality, but can be defined in an application-specific setting.

Java Seal [1] is able to migrate nested seals. On the contrary, our proposed model only supports migration of leaf groups. Additionally, our model does not make any assumption on the programming language, and allows migration of a group only if it sponsors a single thread — the thread itself is not migrated, but it essentially initiates a remote method invocation, i.e. weak migration. On the other hand, our *pause* operation is recursive, and is able to pause recursively a whole hierarchy of groups.

6 Conclusion

In this paper, we have extended Quantum, our framework of resource management to the distributed setting, by introducing explicit remote method invocation (or weak migration) and multiple types of resources. Quantum itself does not provide the accounting mechanism, but it sits on top of such a mechanism to provide a programmatic interface that can be used by resource consumers and providers to program complex distributed applications and their hosting platforms.

Our generic model of resource consumption was successfully applied to many types of resources, including: processor time, wall-clock time, file permissions, write-once objects (such as future placeholders), number of messages, memory and disk usage, migration hops, etc. The same framework is also able to express throughputs and rates, such as message rates, bandwidth, migration frequency, etc.

References

[1] Cyaran Bryce and Jan Vitek. The JavaSeal Mobile Agent Kernel. *Autonomous Agents and Multi-Agent Systems*, 4(359–384), 2001.

[2] Grzegorz Czajkowski and Thorsten von Eicken. JRes: A Resource Accounting Interface for Java. In *Proceedings of ACM OOPSLA Conference*, Vancouver, BC, October 1998.

[3] R. Kent Dybvig and Robert Hieb. Engines from Continuations. *Computer Languages*, 14(2):109–123, 1989.

[4] Matthias Felleisen and Robert Hieb. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science*, 2(4):235–271, 1992. Technical Report 100, Rice University, June 1989.

[5] Jay Lepreau Godmar Back, Wilson C. Hsieh. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, October 2000.

[6] Robert H. Halstead, Jr. New Ideas in Parallel Lisp : Language Design, Implementation. In T. Ito and Robert H. Halstead, editors, *Parallel Lisp: Languages and*

Systems. US/Japan Workshop on Parallel Lisp. Japan., number 441 in Lecture Notes in Computer Science, pages 2–57. Springer-Verlag, 1990.

[7] William A. Kornfeld and Carl E. Hewitt. The Scientific Community Metaphor. *IEEE Trans. on Systems, Man, and Cybernetics*, pages 24–33, January 1981.

[8] Luc Moreau. Distributed Directory Service and Message Router for Mobile Agents. *Science of Computer Programming*, 39(2–3):249–272, 2001.

[9] Luc Moreau and Christian Queinnec. Design and Semantics of Quantum: a Language to Control Resource Consumption in Distributed Computing. In *Usenix Conference on Domain-Specific Languages (DSL'97)*, pages 183–197, Santa-Barbara, California, October 1997.

[10] Luc Moreau and Christian Queinnec. Distributed Computations Driven by Resource Consumption. In *IEEE International Conference on Computer Languages (ICCL'98)*, pages 68–77, Chicago, Illinois, May 1998.

[11] Randy B. Osborne. Speculative Computation in Multilisp. An Overview. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 198–208, Nice, France, June 1990.

[12] José M. Piquer. Indirect Reference Counting: A Distributed Garbage Collection Algorithm. In *Parallel Architectures and Languages Europe (PARLE'91)*, pages 150–165, 1991.

[13] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Paul T. Groth, Gregory A. Hill, Renia Jeffers, Timothy S. Mitrovich, Brian R. Pouliot, and David S. Smith. NOMADS: Toward a Strong and Safe Mobile Agent System. In *Proceedings of the Fourth International Conference on Autonomous Agents*, pages 163–164, Barcelona, Catalonia, Spain, 2000. ACM Press.

[14] A. Villazón and W. Binder. Portable resource reification in java-based mobile agent systems. In *The Fifth IEEE International Conference on Mobile Agents (MA'2001)*, December 2001.

[15] James E. White. Telescript technology: Mobile agents. In Jeffrey Bradshaw, editor, *Software Agents*. AAAI Press/MIT Press, 1996.