# Mobile Computing with Frugal Objects

Benoît Garbinato
Université
de Lausanne
Ecole des HEC
CH-1015 Lausanne
benoit.garbinato@unil.ch

Rachid Guerraoui
Ecole Polytechnique Fédérale
de Lausanne (EPFL)
School of Computer &
Communication Sciences
CH-1015 Lausanne
rachid.guerraoui@epfl.ch

Jarle Hulaas
Ecole Polytechnique Fédérale
de Lausanne (EPFL)
School of Computer &
Communication Sciences
CH-1015 Lausanne
jarle.hulaas@epfl.ch

Ole Lehrmann Madsen
Aarhus University
Department of
Computer Science
DK-8200 Aarhus N
ole.l.madsen@daimi.au.dk

Maxime Monod
Ecole Polytechnique Fédérale
de Lausanne (EPFL)
School of Computer &
Communication Sciences
CH-1015 Lausanne
maxime.monod@epfl.ch

Jesper Honig Spring
Ecole Polytechnique Fédérale
de Lausanne (EPFL)
School of Computer &
Communication Sciences
CH-1015 Lausanne
jesper.spring@epfl.ch

## ABSTRACT

This paper introduces a simple event-based model for programming resource-limited mobile devices. Its originality lies in its *adaptability* and *frugality*. With this model, a program consists of a set of distributed frugal objects, called FROBs[1], that communicate through typed events and that may dynamically adapt their behaviors based on resource availability.

To achieve frugality, a FROB relies on two resource awareness mechanisms: the ability to express resource needs before executing an action, and a logical time-slicing model to control resource-consuming tasks (typically loops). Resource needs are expressed via a resource model describing CPU, memory, bandwidth, etc., whereas logical time-slicing makes it possible to remove loops completely. Adaptability is then achieved by dynamically changing the set of event types a FROB can accept and the set of corresponding actions for treating them. In particular, a FROB can shift its capabilities in case of contention, or augment its set of actions at runtime by downloading code from some remote FROB.

We illustrate the use of our model on a concrete peer-to-peer audio streaming example, showing in particular how it supports graceful service degradation due to shortage of resources at runtime. We also evaluate our approach via a first prototype of our model based on Sun's Java KVM targeted at resource constrained devices.

[1] *Random small things*: The Free On-line Dictionary of Computing, http://www.foldoc.org/

## 1. INTRODUCTION

As millions of mobile devices are being deployed to become ubiquitous in our private and business environments, the way we do computing is changing. We are moving from static and centralized systems of wire-based computers to much more dynamic frequently changing distributed systems of heterogeneous mobile devices. These devices, which might be embedded, are typically communication capable, loosely coupled, and constrained in terms of resources available to them. In particular, it is expected that many of such devices be frugal in terms of processing power, storage and bandwidth, for these may or not be available, depending on the mobility pattern and the solicitations. The devices might communicate using wireless LANs, satellite links, cellular networks, or short-range radio links.

The software components of mobile devices are usually supposed to automatically discover each other on the network and join to form ad-hoc communities enabling mutual sharing of each others functionalities by offering and lending services.

In an ever changing environment, the resource awareness of the components and their ability to provide a high degree of adaptability are paramount to their operation. Adaptability means here, for a component, the ability to adjust the level of service it offers following changes in the resource availability [31, 17].

To illustrate one aspect of adaptability through a simple example, imagine a mobile service offering to stream some audio upon request to interested mobile clients. To start with, the audio service might be able to offer the first few interested clients an audio stream with 128 kilobits high-

quality audio. If more interested audio clients show up in the surrounding and request the audio stream, the audio service might be forced to reduce the quality of the stream to cope with the demand. This service degradation might occur a number of times before the service might actually decide to reject additional requests until it has more available resources.

Despite the importance of adaptability and resource awareness, very few systems address these issues in a unified coherent framework, as detailed in [33], and as we briefly summarize in the related work section. Most systems that are marketed for mobile computing are scaled-down descendants of programming models designed with coarse-grained static applications in mind. These include specific variants of the Java [19] and .Net frameworks [28].

We argue that the principles that should drive the design of a computing model for frugal mobile devices lead instead to fundamentally revisit that filiation. Three principles seem of primary importance:

1. *Exception is the norm.* The distinction between the notion of a main flow of computing and an exceptional flow (i.e., a plan B) is rather meaningless in mobile environments. As discussed above, the software component of a device should adapt to its changing environment and cannot predict the mobility pattern of surrounding devices or even the way the resources on its own device will be allocated. The fact that something exceptional is always going on [33] calls for a computing model where several flows of control can possibly co-exist, or even be added or removed at run time.

2. *Resources are luxuries.* Just like it is nowadays considered normal practice that a software component is able to adjust to specific changes on some of its acquaintance components, and react accordingly, we argue for a computing model where the components can react to the shrinking of local resources, as well as to changes to their communication capabilities. This calls for a computing model where the components are made aware of the resources they use, and can use them in a frugal way. The fact that resources are luxuries also mean that certain greedy programming habits, such as *loops*, *forks* or *wait* statements, should be used, if at all, parsimoniously.

3. *Coupling is loose.* Many distributed computing models have been casted as direct extensions of centralized models through the *remote procedure call* abstraction (e.g. [32, 29, 38]). This abstraction aims at promoting the porting of centralized programs in a distributed context. Clearly, such abstraction makes little sense when the invoker does not know the invokee, or even know whether there is one at a given point in time. Some of the extensions to the abstraction, including futures [35] (also called *promises* [26]) only address the synchronization part of the problem. Mobile environments rather call for anonymous and one-way communication schemes.

Needless to say, devising a robust computing model that, while obeying the above principles, remains simple to comprehend yet implementable on small devices, is rather challenging. The aim of our work is precisely to face that challenge, through a candidate computing model based on FROBs.

In our model, a program is modeled as a set of reactive frugal objects, called FROBs. FROBs are units of distribution that communicate through events and adapts the set of event types they can handle and the way they handle them at any point of their computation according to the resources they have at their disposal. The FROBs are deployed and executed on a FROB runtime, which provides a common infrastructure for communication and resource awareness.

## 1.1 Typed Events, Conditions and Actions

In the FROB model, computing is triggered by *typed events* that regulate the anonymous and asynchronous communication between FROBs. A FROB can specify, at any point of its computation, (a) the type of events it can process, (b) its *actions*, i.e., how the events should be processed, and (c) its *conditions*, i.e., under which circumstances (in terms of resource availability) the actions should be processed.

The set of typed events that a FROB can process acts as its *interface.* Interfaces are used by the runtime to dispatch events to the FROBs within a device, or even by a clusterhead in an ad-hoc network to route events within a certain geographical zone[2], see Figure 1.

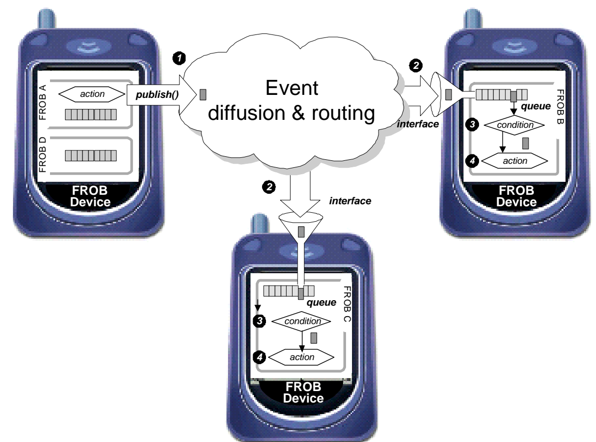



**Figure 1: Event-based Interacting FROBs**

Once an incoming event matches the interface of a FROB, it becomes a candidate to trigger a computation within that FROB.

When an incoming event actually triggers a computation depends of the conditions expressed by the FROB. Conditions provide the means for a FROB to specify resource-based scheduling strategies, as well as priority-based concurrency

[2]Although not covered in depth in this paper, these interfaces along with the typed events also provide means of integrating FROB-based systems with traditional back-end systems.

control schemes. The conditions of a FROB are ordered and are matched against events accordingly.

An action is a sequential unit of computing that is executed based on some event. The action might modify the FROB's state, as well as its behavior, by adding or removing event types to/from the interface, conditions and actions.

Each condition is associated with an action through a *trigger*, which is an atomic unit of computing. The atomicity of the condition-action pair significantly simplifies execution and concurrency control. Although reachable to the runtime, the triggers, conditions and actions are considered part of the internals of a FROB – its state – and are invisible to the outside (particularly to other FROBs).

## 1.2 Adaptability based on Resource Awareness

The FROB model promotes adaptability based on resource awareness in the system on several accounts. In order to handle certain events, the FROBs actively have to request necessary resources; they have to request the runtime to *give* some resources to them. This is necessary in order for the FROB to have enough resources available to execute the action. In addition, the FROBs are passively informed by the runtime about critical changes in the resource levels to which they can choose to respond.[3] Since the runtime manages resources shared between multiple FROBs, over time the demand for resources might change in such a way that the resource availability gets constrained. In this case, the runtime thus requests the FROBs to willingly *give up* some of the shared resources.

## 1.3 Dynamic Code Replacement, Distribution and Mobility

As mentioned above, FROBs are adaptive to changes in resources. This adaptability is expressed in terms of changing its interface or changing internal behavior. Whereas the former involves adding or removing event types to the already existing interface, the process of adapting behavior is typically done through replacement of code, which in inherently supported in the FROB programming model. The act of changing interface or changing internal behavior through behaviour replacement is done dynamically during runtime.

The code used to replace existing code in a FROB might be loaded locally or received from another possibly remote FROB. The FROB model provides means for the FROBs to exchange individual parts of their code rather than the entire code, which can then be used to upgrade code in a fine-grained manner during runtime.

The FROB model also allows mobility as part of the adaptation to changes in resource availability. As such, FROBs can adapt to resource changes by requesting the runtime to be migrated to another devices, where it can better exploit remote resources. If both the source and destination runtime agrees to this migration, the runtime will migrate the running FROB.

[3]Assuming that the FROBs have subscribed to these event types in their interfaces.

## 1.4 Prototype Implementation

We have implemented a prototype of the FROB runtime on top of the Java KVM virtual machine, which as part of the J2ME CLDC/MIDP [36, 37] platform is targeted at resource constrained mobile devices. The current prototype implementation of the FROB runtime adds a memory overhead of approximately 117 kilobytes or roughly 5% to the J2ME CLDC platform version 1.1.[4]

Using this FROB runtime, we have implemented a demanding scenario involving ad-hoc audio streaming between mobile devices. This scenario is especially challenging from an adaptability point of view. It enables FROB-based clients to discover peer audio providers and to request permission and obtain the functionality needed to play the provided audio stream.

It should be stressed that although this paper uses Java, or rather a deliberate subset of the language constructs, for the FROB runtime implementation and particularly the scenario, it is only as a prototyping platform. As such, this does not imply that the FROB model is bound to Java nor any other particular language or execution platform.

## 1.5 Contributions

To summarize, the contributions of the paper are a candidate event-based computing model, which promotes resource-based adaptability through a notion of typed yet dynamical (i.e. which can change during runtime) interface, as well as the ability to change behavior through code distribution, replacement and migration.

The rest of the paper is structured as follows. Section 2 overviews related work, and positions the FROB computing model with respect to alternative distributed computing models. Section 3 describes the main concepts underlying the FROB model and gives an overview of the scenario used throughout the paper. Section 4 describes the resource awareness scheme underlying the FROB model as well as aspects of resource-based adaptability. In that section, we also exemplify these aspects through extensions of the audio streaming scenario. Section 5 evaluates the FROB model by also describing the prototype runtime (based on the small Java J2ME CLDC platform) on which FROBs are currently executed, along with some initial measurements on memory requirements. Finally, Section 6 gathers some final remarks.

# 2. RELATED WORK

As we pointed out in the introduction, the computing models currently marketed by the industry for building mobile applications, such as Java CLDC [36] and .Net Compact Framework [28], are descendents of programming models used to build traditional applications for more static environments. Unfortunately, neither the models, nor their runtime support, provide the constructed applications with adequate ability of combined adaptability and resource awareness. Although there have been attempts to address aspects of adaptability in certain variants of such systems, such as [4, 41], these only focus on very coarse-grained levels of adaptability. The focus is solely on connectivity: a subset of the

[4]Running MIDP version 2.0 augmented with additional packages as described in Section 5.

centralized programming model and a trimmed down version of the runtime system are provided.

SEDA [42] is an event-based approach which focuses, like we do, on designing systems that behave gracefully even under severe load; however, whereas SEDA proposes a rather fixed architecture for Internet servers, FROBs aim at representing a more general-purpose computing programming model.

There is no broadly accepted programming model for resource management, and, a fortiori, resource awareness. Several prototypes have been proposed, using Java as execution platform, such as the Aroma VM [40], KaffeOS [5] and the Multi-tasking Virtual Machine (MVM) [11], which suffer from their lack of portability, and therefore applications running on such platforms may not be deployed throughout a large variety of devices, including embedded systems. JRAF2 [7] is a resource management framework that is portable, since it is based exclusively on bytecode instrumentation of applications, middleware and runtime support [22]. JRAF2 has been tested in many settings, from grid computing to desktop applications, including, with some limitations, embedded devices.

Another important obstacle that researchers in resource management are facing, is that efficient resource management also requires appropriate isolation (such as Unix processes) between the supervised entities, in order to prevent unwanted interferences such as deadlocks when a misbehaving entity is sentenced to be throttled or killed. Whereas Java does not yet provide such a facility, research is currently pursued in that direction [12].

For very resource constrained devices, the most notable project is TinyOS [21], which is centered around sensor networks. Like TinyOS, the FROB model is based on an event-based programming model, which resource-wise is a cheap alternative to multi-threading systems that are expensive in terms of stack management and over-provisioning of stacks, as well as locking mechanisms [14]. TinyOS provides resource awareness in the sense that it has been designed to be conservative in its resource consumption; but it cannot in any way adapt to changes in the levels of resource by changing behavior. Compared to TinyOS where the code, once linked and deployed on a device, cannot be changed, the FROB platform provides fine-grained means for non-interrupted runtime code replacement.

The Maté [25] project addresses this issue by providing a virtual machine for TinyOS devices on which very small blocks of code, *capsules* of 24 instructions, can be replaced. However, limited to small blocks of code, this solution is still very inflexible compared to the FROB model.

Other projects, such as [14, 30], have also addressed adaptability – though still somewhat inflexible – with predefined service levels and infrastructure responsibility to actually initiate the possible changes.

The FROB programming model has its roots in the seminal work of Dijkstra on guarded commands [13] and its derivatives [6]. The underlying idea is to divide the programming into a set of atomic actions protected by predicates. A predicate determines the exact conditions under which a certain action can be executed. In the FROB context, these predicates, called *conditions*, typically evaluate the availability of resources that are needed to perform a given action, and help express priority-based concurrency control strategies. Interestingly, condition/action pairs themselves are guarded by an interface of event types that the FROB is willing to accept at any point in time of its computation.

Unlike many distributed computing models [23, 3, 43], only one action at a time is executed by a FROB, and actions do not contain synchronization statements. Besides the underlying issues of thread and CPU management, such statements (e.g., *wait*, *fork/spawn*) break the atomicity of the actions and significantly complicate code upgrading, concurrency control, and resource-based reasoning. In particular, *remote procedure calls*, be completely synchronous, or semi-synchronous through the use of futures [35] or promises [26], are precluded within FROB actions. If needed, they are programmed through events and conditions across several action executions. In this sense, our model is close to the actor model [20, 1], and more precisely its actorspace [2] variant with anonymous event-based communication (itself inspired by [24]). There are however three important differences. First, whereas an actor is an immutable object (state changes are achieved through the creation of new actors - *become* statement), a FROB is on the contrary expected to change its state and behavior. This has a direct impact on the programming style but also, and may be more importantly, on the runtime management of object identities and memory allocation. Second, the notion of (internal) *condition*, which is an inherent part of our model, provides a flexible way to devise concurrency control and resource-based scheduling strategies. The development of such strategies using only *send* and *become* statements of the actor model clearly leads to the explosion of a program into many independent actors, which hampers resource-based reasoning and code upgrading. Third, FROBs have a type oriented notion of interface. At any point in time, the set of event types that a FROB can handle is precisely defined, and this facilitates code reuse and prevents casting errors.

Like Emerald [23], the FROB computing model supports migration of running processes. However, unlike Emerald, the FROB computing model does not support migration the process' thread. The FROB computing model is inherently threadless, and thus maintains a loose coupling between the actions and the thread executing them. The threads are assigned to the execution of actions by the runtime in a time-slicing scheme. Given this time-slicing scheme, and the fact that, within any FROB, only one action at a time is executed, the checkpointing of the runtime state of a running process between two action executions is straightforward.

The FROB computing model is inherently anonymous and asynchronous, without any global clock concept, unlike in purely synchronous models [9], or even hybrid models like [10].[5] Processes are notified of events following a type-based publish/subscribe interaction pattern [15, 16] (unlike most concurrent programming languages [23, 3, 43]) and do not need

[5]Of course, more coupled forms of communication abstractions can easily be developed on top of the basic FROB communication infrastructure.

to pull a shared space for events they are willing to treat (unlike in Linda [18]).

# 3. THE FROB COMPUTING MODEL

As illustrated in Figure 2, from the computing viewpoint, a FROB basically consists of: (1) an *interface* made up of event types, (2) a FIFO ordered queue of *events*, and (3) an ordered list of *triggers* (each made of a *condition* and an *action*).

The interface and triggers (including the conditions and actions) are contained in a data structure called a *dictionary* (see (4) in Figure 2). The dictionary also references other parts of the state and code of the FROB – in all cases typed entities. In this sense, the dictionary *represents* the FROB. Since slots are always *named*[6] and their content must be referenced through this name, this level of indirection makes it possible to redefine behavior by changing their content. In fact, all entries in the list of triggers (i.e., the individual conditions and actions) are names referring to slots in the dictionary where the actual condition and action are located.

The event queue of the FROB (2) is not contained in the dictionary and is under the sole control of the runtime, i.e., the FROB has no direct access to it and its only way to consume events is by defining adequate triggers. This enforces a declarative model of programming with multiple flows of control.

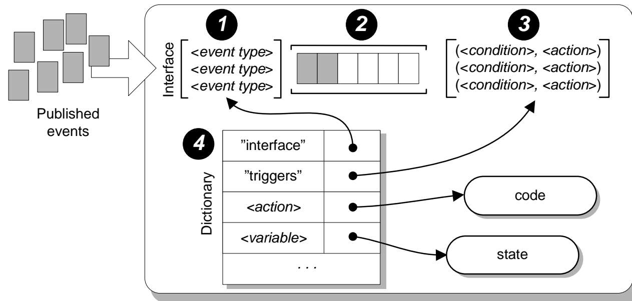



**Figure 2: Programmer's View of a FROB**

The runtime places published events into the event queue of a FROB if they match at least one of the event types in its interface. If there are events in the queue, the runtime evaluates each event against all triggers of the FROB, in the order defined by the event queue and the trigger list, and executes the first trigger that evaluates to *true*. In this context, evaluating a trigger actually means evaluating its condition, whereas executing a trigger means executing the code corresponding to its action. This code is accessible as a named slot in the dictionary.

## 3.1 Events

An *event* is the basic entity to which FROBs react, i.e., an event might cause an action in a FROB to be executed. The events serve as communication units between multiple FROBs, whether deployed on different devices or on the same device. The events are typically *published* by the FROBs themselves, but might also be generated by the runtime following some internal event.

$FROB$ *denotes a frugal object*
$e$ *denotes an event*
$E = \{e_1 < e_2 < ... < e_n\}$ *denotes a queue of accepted events*
$E \in FROB$ *and is denoted* $FROB.E$

**Figure 3: Events on the Event Queue**

All events received and *accepted* by a FROB are put on the FIFO ordered event queue associated with the FROB (in Figure 3) for further processing.

An event can have an expiration time associated. The timeout defines the maximum time that an event may live before having to be processed by a FROB. After this time, the event expires and (a) is thus not disseminated any more, and (b) the event is removed from the event queue of the FROB after which the runtime notifies the FROB through a system event describing that the original event expired. It is optional whether the FROBs want to react to this system event.

## 3.2 Interfaces

An *interface* expresses *what* typed events a FROB will react to. For an incoming event to be added to the FROB's event queue, the FROB must have expressed interest in the event. This is done through the interface, which consists of a set of *event types*. The event types can be defined by a name and possibly a predicate expression.

If a FROB is interested in an event, it *accepts* it and puts it on its event queue; otherwise it ignores it. The properties and operations of the interface are seen in Figure 4.

$i$ *denotes an event type*
$I = \{i_1, i_2, ..., i_n\}$ *denotes a set of registered event types on the interface*

$dic$ *denotes a dictionary*
$dic \in FROB$
$I \in dic$ *and is denoted* $FROB.I$

*operations on* $I$ :
$I = \{i_1, ..., i_n\} \vdash I.add(i_x) \Rightarrow I = \{i_1, ..., i_n, i_x\}$
$I = \{i_1, ..., i_j, ..., i_n\} \vdash I.remove(i_j) \Rightarrow I = \{i_1, ..., i_n\}$

*event type matching* :
$e \ : \ event \vdash i : e \rightarrow \{true, false\}$

**Figure 4: Interfaces and their Operations**

The acceptance of an event is decided on the result of matching the event against the interface of the FROB. The evaluation of the event against the interface is done *externally* – based only on the received event – and thus independently of the internal state of the FROB. The procedure involves matching based on (a) the *event type*, and also possibly (b) *event content*. Thus, if the event has a type not represented in the interface, then the event does not match and is ignored. On the contrary, if the event has a type represented in the interface, the matching continues on the content of

[6] In Figure 2 illustrated by names in quotes (for specific names) or enclosed in <> (for *some* name).

the event, if required. To perform the match on the content of the event, it first has to be deserialized after which the predicate-based expression can be matched against the contents of the event.

Using these interfaces, two FROBs can model a point to point communication scheme on top of the underlying communication scheme. This can be done through the specialized event types with appropriate predicate expression matching an address name, which the events sent between the FROBs include.

## 3.3 Triggers

A *trigger* represents the common interface between an event and an action. The FROB represents this interface through an ordered list of triggers, which the FROB through the dictionary can manipulate at any time during runtime. The ordering in the list has the effect of prioritizing triggers during their inspection. The properties and operations of triggers are depicted in Figure 5.

$$
\begin{array}{l}
t\ denotes\ a\ trigger \\
T = \{t_1 \lhd t_2 \lhd ... \lhd t_n\}\ \ denotes\ an\ ordered\ list \\
\qquad\qquad\qquad\qquad of\ triggers \\
T \in dic\ and\ is\ denoted\ FROB.T \\
\\
operations\ on\ T: \\
T = \{t_1 \lhd ... \lhd t_n\} \vdash T.add(t,i) \Rightarrow \\
\qquad\qquad T = \{t_1 \lhd ... \lhd t_i \lhd ... \lhd t_n\} \\
T = \{t_1 \lhd ... \lhd t_i \lhd ... \lhd t_n\} \vdash T.remove(i) \Rightarrow \\
\qquad\qquad T = \{t_1 \lhd ... \lhd t_n\}
\end{array}
$$

**Figure 5: Triggers and their Operations**

A trigger is a logical abstraction that binds a *condition* with an associated *action*. Since conditions and actions are named entries in the dictionary, they can be associated through multiple triggers concurrently.

## 3.4 Conditions

A *condition* is used to express *when* the action of the trigger should be executed, that is, under which resource-based circumstances. A condition is a combination of predicates expressed in terms of the event, internal FROB state (represented by the dictionary), and system properties provided by the runtime (i.e. available computing resources etc.). A condition is bound to events of a given type.

A condition – or rather its predicate expressions – is evaluated in the context of a given event, which was accepted by the FROB's interface. During this evaluation, the condition cannot change the internal FROB state – only query values in the dictionary. Upon evaluating a condition to *true* it causes the action associated in the trigger to be executed. An event, which causes a condition to evaluate to true, is said to be *appropriate* for processing.

## 3.5 Actions

An *action* is a piece of code – located in a named slot in the dictionary – which defines *how* the FROB should deal with an accepted event following the successful evaluation of the associated condition. This is depicted in Figure 6.

The action is executed in a context consisting of the first *appropriate* event (which is removed from the event queue), and the dictionary of the FROB. Thus, the action can read, modify and write values in the dictionary. Since the dictionary contains everything that defines the FROB, an action has the ability to completely redefine the FROB behavior – even the action itself; dynamically upgrading a FROB can for example be achieved that way.

$$
\begin{array}{l}
c\ denotes\ a\ condition,\ element\ of\ the\ set\ C \\
a\ denotes\ an\ action,\ element\ of\ the\ set\ A \\
\\
t\ is\ a\ trigger \Leftrightarrow t = (c,a) \in C \times A \\
R\ denotes\ the\ resource\ domain \\
event\ e,\ dictionary\ dic,\ R\ r \vdash c(e,dic,r)\ : \\
\qquad\qquad E \times dictionary \times R \rightarrow \{true, false\}
\end{array}
$$

**Figure 6: Conditions and Actions**

The runtime guarantees that the action executes atomically with respect to the FROB, i.e, two actions of the same FROB cannot execute concurrently. In addition, since FROBs are encapsulated entities, they cannot share state (i.e. entries in the dictionaries). This combination eliminates the need for synchronization on entries in the dictionary.

An action might publish events to other FROBs (or to itself), using the non-blocking `publish` primitive provided by the runtime, which provides best-effort delivery. Events can be published to all interested FROBs (by specifying no explicit receiver) or explicitly addressed to a specific receiving FROB (including itself) using some event types.

## 3.6 FROBs

A FROB contains all entities described above: a dictionary, the interface, the triggers along with other state and code, and its queue of events. FROBs are isolated entities, which cannot reference each other directly. Instead, to collaborate they publish events.

### *3.6.1 Life-cycle Properties*

Once deployed on a runtime, the FROB is then executed and active. A FROB eventually becomes *idle* when its queue of events is empty; there is nothing for the FROB to react to.

If a FROB contains an empty set of triggers, or if the FROB's interface and event queue are empty, and the FROB is not executing an action, it is declared dead and garbage collected by the runtime as depicted in Figure 7.

$$
\begin{array}{l}
FROB.E = \varnothing \Leftrightarrow FROB\ is\ idle \\
FROB.T = \varnothing \vee (FROB.I = \varnothing \wedge FROB.E = \varnothing) \Leftrightarrow \\
\qquad FROB\ is\ dead
\end{array}
$$

**Figure 7: FROBs Life Properties**

Having only an empty interface is not sufficient to declare a FROB dead. There still might be some events in its event queue that the FROB can react to when the conditions are satisfied, which might cause new event types to be added to the interface.

### 3.6.2 Code Distribution Aspects

As mentioned earlier, FROBs can publish events to each other. These events typically contain raw data. However, FROBs can also send code to each other such as actions or even entire dictionaries.

The sending of entries, such as actions, between FROBs enables one FROB to inherit partial functionality from another one by, for instance, integrating the received action into its own dictionary. This is very useful in order to adjust say the level of service between two collaborating FROBs, or correct a faulty piece of code.

In contrast, the sending of whole dictionaries allows for distribution of complete and isolated functionality. Since the dictionary is basically the FROB, a FROB can be created from any dictionary.

## 3.7 The FROB Runtime

The runtime supports the ability of a FROB to spawn another FROB (on the same runtime) from a dictionary. The provided dictionary (including its entries) is cloned to prevent multiple FROBs from sharing data. Being able to spawn a FROB from a dictionary, it is possible to distribute encapsulated functionality between runtimes, which is useful for instance through a proxy or when migrating a FROB.

### 3.7.1 Receiving and Accepting Events

An event is received by an individual FROB by adding it to the end of the FIFO ordered queue, which conceptually is associated with each deployed FROB, where it will await further processing. Figure 8 depicts the runtime operations on the event queue.

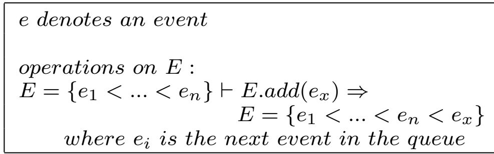



**Figure 8: Runtime Operations on the Event Queue**

When the runtime receives an incoming event, it checks if any deployed FROBs are interested in the event, i.e., any interfaces containing event types matching the incoming event have been registered. If the runtime finds a FROB having registered an event type in its interface that matches the incoming event, a copy of the incoming event is then stored in the FROB's queue of accepted events for further processing as seen on Figure 9.

$$\bigvee_{i \in FROB.I} i(e) = true \Rightarrow (FROB.E).add(e)$$

**Figure 9: Accepting incoming Event**

### 3.7.2 Processing Accepted Events

The runtime continuously processes the event queue of the deployed FROBs through iteration as long as the queue is non-empty. During each iteration, at most one event at a time can be removed from the queue – the first *appropriate* event found in the queue, if any, for which a trigger evaluates to true. This processing is illustrated in Figure 10.

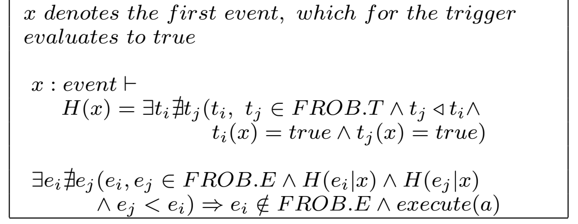



**Figure 10: Processing the Queue of Events**

When an event is removed from the queue, and the action executed with the event, the iteration restarts from the beginning of the queue.

## 3.8 A FROB Scenario

To demonstrate the viability of the FROB model, we have designed and implemented a scenario on audio streaming between mobile devices. The scenario provides several challenges in terms of adaptability to changes in resources. As such, the scenario includes resource adaptability following change in connectivity (the service that one relies on appears/disappears), workload (the number of clients requesting audio streaming) and resilience (in terms of having resources to provide audio data in a steady stream).

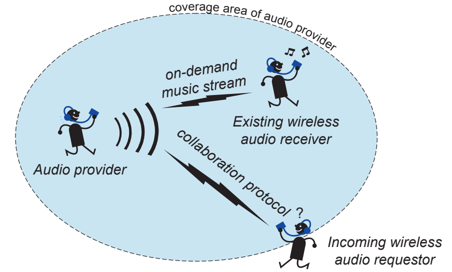



**Figure 11: The Audio Streaming Scenario**

The scenario involves multiple devices running FROBs; one FROB provides audio streaming capabilities to a number of interested client FROBs wanting to play the audio stream it provides, as illustrated by Figure 11.

The devices are all connected to some wireless network on top of which they form an ad-hoc network. Once the networking capabilities have been set up, the server and client initiates mutual discovery. Having discovered each other, the FROBs start negotiation for collaboration, i.e. the client FROB requests the server FROB for permission to use its service – receive the audio stream. Having gained permission, the client receives the capabilities to decode the audio stream after which the audio streaming starts.

The scenario also exposes problems inherent to ad-hoc net-

works and resource aware devices/runtimes, like, for instance, how the audio provider should react as the number of requesting audio clients is growing (can the same quality of service be provided, when and how should the service then be degraded?) etc.

# 4. FRUGALITY AND ADAPTABILITY

Key to the FROB model is the built-in support for frugality in terms of available resources as well as its various support forFROBs to adapt to changes in their environment, including the resource availability. This support is present at different levels in the model: the language level and in the runtime.

## 4.1 Logical Time-Slicing Model

The FROB computing model promotes a programming style where long running procedures are split up into small, short-lived event-based execution units (*actions*), which, once executed by the runtime, are allowed to run to completion. The resource requirements of these individual actions are thus limited in terms of actual resource amount needed and required duration.

These actions are executed based on accepted events. Events in some FROB's queue are thus interpreted by the runtime as requests to ultimately get some execution time. The runtime can thus be seen as providing a kind of logical time-slicing model; a queued event represents a request for some time-slice, which is granted when the action consuming that event is executed.

This scheme of small, short-lived execution units is also promoted by the fact that the FROB programming model precludes the use of loops, forks, and synchronization primitives in the actions. This, for instance, prevents the execution of an action from thread monopolizing the CPU. In addition, since the computing model defines no blocking primitives, a FROB has no way to compromise liveness either.[7]

In a way, the computing model requires loops to be explicitly unfolded. As such, internal *loops* are programmed using the event system that explicits the need for more CPU cycles to the runtime. For this purpose, the FROB has to publish an event for which it has specified the appropriate event type as part of its interface. So, although loops and recursive calls are syntactically absent from our model, iterative processing can still be achieved, but in a way that prevents starvation or violation of liveness. Indeed, the runtime benefits from an explicit control point during the iteration, in the form of an event representing the time-slice request for the next iteration.

Similarly, fork or wait statements are achieved through events that systematically yield the control to the runtime. Besides concurrency control and resource-based reasons, these explicit control points also make it easier to checkpoint, migrate and manipulate (i.e. behavioral change) the FROB.

In the audio streaming scenario, the whole scenario has been implemented using short-lived actions. For instance, to perform the actual streaming, we have implemented an action in the audio stream provider that reacts to an event-based request to send the next chunk of audio data. Once this action is executed (and the event is consumed), the next piece of the audio data is read and published to the requesting audio client.

## 4.2 Requesting Action Resources

Core to observe frugality is to react to a given event in a resource aware manner. Our model promotes resource-aware programming by allowing conditions to be expressed in terms of resource requirements of the associated action.

The conditions represented as predicates are evaluated by the runtime, as depicted in Figure 12, which based on internal resource monitoring and control, will determine if the conditions can be satisfied or not, and in the former case reserve and provide the resources to the FROB. The resource model defined in the runtime is non-restrictive in the sense that it does not prevent an action in a FROB from consuming more resources than potentially specified by its associated condition.[8]

$$\begin{array}{l}
R \text{ denotes the resource demand domain} \\
CPU \text{ denotes the } CPU \text{ rate requirement} \\
Memory \text{ denotes the } Memory \text{ requirement} \\
Bandwidth \text{ denotes the } Bandwidth \text{ requirement} \\
r = (c, m, b) \in CPU \times Memory \times Bandwidth \\
\qquad\qquad \text{denotes a resource requirement} \\
\\
Resource\ request\ predicate: \\
r \ : \ CPU \times Memory \times Bandwidth \rightarrow \{true, false\}
\end{array}$$

**Figure 12: Resource Definition**

The resources managed by the FROB runtime include those tied to the physical platform such as CPU rate, memory and bandwidth. By specifying the resource requirements, the runtime guarantees – once the conditions have been satisfied – that the required resources will be available to the action throughout its execution. For CPU, this means that the runtime guarantees that, once the action is executed, it is on average given enough CPU cycles to execute at the requested rate. This is critical to actions with software-based real-time aspects. Likewise, for memory, the runtime guarantees that no failures due to memory allocations will occur. For bandwidth the resource model can guarantee little in terms of absolute guarantees due to external events stemming from the loosely coupled environment of the device. Instead, the guarantee must be expressed as relative to the connectivity present, if any. Figure 13 illustrates a condition in the dictionary expressing resources.

If a condition specifies no resource requirements, the run-

[7]One alternative considered for a practical application, one might want to allow loops/recursion yet still prevent this monopolization and ensure liveness by defining a maximum execution time for an action. After this maximum execution time the runtime would terminate the uncompleted action. This would however require transactional execution of the action to prevent the state from being undefined following a termination.

[8]However, there is nothing in the FROB model that fundamentally prevents the use of a more restrictive resource implementation where each resource allocation and deallocation made in an action has to be explicitly approved by the runtime.

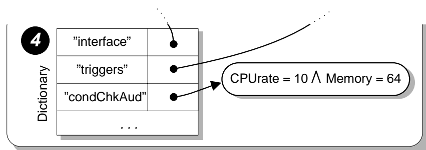



**Figure 13: Requesting Resources in Condition**

time will execute the associated action in a best-effort way, meaning that it cannot guarantee that the action will be performed at for instance a given CPU rate, and thus might be interrupted for short periods, nor guarantee that the required memory is available, which thus might cause memory related runtime errors.

Using the ability to express resource requirements for an action, a FROB might for instance define two triggers that can handle the same event type, where the condition of the first trigger expresses how to handle the event in the best-case (where all its required resources are available), and condition of the the second expresses what to do, if the required resources cannot be provided by the runtime.

In the above audio streaming scenario, this ability to behave in a frugal manner by performing certain actions only when required resources are available, is heavily used. As an example, we use it in the audio receiver to play the actual audio packet received in the event. By requesting these resource requirements, the audio receiver knows that, once the action starts executing, the runtime guarantees the resource requirements which allows the action to run to completion without any resource related errors.

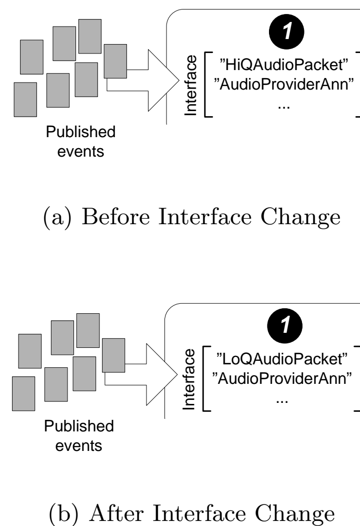
## 4.3 Reacting to Resource Notifications

The runtime constantly monitors resource consumption of the FROBs that are deployed. This information is accessible to the FROBs via an interface in the runtime. From this interface, the FROBs can query the current resource availability on which they can base decisions.

However, the resource model in the runtime is additionally pro-active in the sense that it notifies the deployed FROBs if major changes in resource availability occurs. The simplicity of our model results from the consistent use of typed events and conditions to control the life cycle of a FROB. For example, no exception mechanism is provided: special events are used instead, e.g., to indicate a shortage of resource.

It is optional for the FROBs to react to these resource related system events. However, it is in the best interest of all FROBs to collaborate with the runtime, as the runtime might otherwise be forced not to allocate time-slices to a given non-collaborative FROB.

By responding to these system events, the FROBs might be, for instance, informed that the runtime is running low on memory and thus asked to free as much memory as possible, which it, for instance, might do by degrading the level of service that it offers. When a FROB changes its level of service, it typically involves changing its internal behaviour, but also its interface; that is, the event types it is interested in receiving as depicted in Figure 14. A FROB may react to many such resource notification events during its lifetime, and thereby either adjust its provided level of service up or down.




(a) Before Interface Change

(b) After Interface Change

**Figure 14: Changing Interface of a FROB**

Continuing the above scenario, such resource notifications may both affect the audio provider and the audio client. For instance, if the audio provider receives a resource notification event from the runtime telling that the level of available resources is low, the audio streaming FROB may initiate a service level degradation. Such degradation might involve stopping to respond to certain requests and instead react to others. However, since the audio clients are all adjusted for the actual level of service provided by the audio provider, the audio clients (or possibly only part of them) might be forced to adapt to this new level of service too. Thus, the consequence of a service degradation following such a resource notificiation might disseminate to the audio clients too. On the other hand, if a single audio client receives a resource notification from its runtime, that it needs to lower its resource consumption, the audio client might ask the audio provider to adjust as well. However, in this case, the audio provider would only degrade the level of service for this single audio client.

The dynamic interface in itself also provides means for resource awareness. The event types defined in the FROB's interface basically express which events the FROB can handle, that is, among other things, should be capable of deserializing. This enables the FROB model to have a fine-grained notion of deserializers. So, rather than having a general deserializer capable of deserializing all possible events, the FROB model loads a deserializer per event type in the interface, and thus only spends memory resources on loading the capabilities it really needs. Any deserializers no longer needed (i.e. no event types in the interface) are thus unloaded from the runtime to free resources.

## 4.4 Adapting through Code Distribution, Replacement and Migration

Behavioural change comes either in the form of replacing a FROB's existing behaviour or by distributing new capabilities to the FROB. Code distribution is an efficient way to distribute the code representing this behavioural change between FROBs. As mentioned earlier, individual pieces of code (i.e., actions) representing small pieces of functionality or some complete functionality, provided in a dictionary, might be distributed between FROBs.

In the above audio streaming scenario, these capabilities are used to distribute new functionality in order for the requesting audio client to be able to read and decode the stream distributed from the audio stream provider. This new functionality is distributed to the audio client by the audio provider as part of their collaboration negotiations. This functionality, represented by a dictionary, is, once received, spawned as a separate FROB on the runtime of the client. Once running and streaming audio, the individual actions in the dictionary of this FROB might be replaced. For instance, if the audio provider degrades its level of service to an extend, which requires parts of the decoding functionality running in the audio client to be replaced. In this case, the audio provider would simply send the audio client the new action(s) to be installed.

Another aspect of adaptability is related to migration of code. The FROB runtime and infrastructure provides means for a FROB to request its runtime to be migrated to another runtime. Such a request might be initiated by the FROB having received a resource notification from the runtime that, for instance, the battery level is getting so low that the device cannot continue operations for a longer period.

## 4.5 Resource Aware Runtime

The adaptability in the form of code distribution, mobility and replacement is typically motivated based on resource availability reasons. However, the FROB model does not mandate the support of all these forms of adaptability in the runtime. Instead, the FROB model promotes a flexible form of adaptability, where a FROB can adapt on different levels depending on the capabilities on the device on which the FROB is deployed. For instance, some devices might be specialized for a single application and thus not support migration of FROBs in any way.

In addition to just monitoring resource consumption and request changes in resource usage from the FROBs, the runtime itself is also obliged to observe frugal behavior and act accordingly in order to prevent resource related runtime errors from occuring within the runtime.

The issue of resource consumption in the runtime is particularly related to the collaboration between the runtimes on different devices, and thus the exchange of data such as events and code, which at some point in the life-time of the runtime might cause problems with resources. For instance, the runtime might receive an event, which due to its size combined with the available resources cannot be deserialized, or the runtime might receive events faster than a FROB can consume. In such cases, the runtime might choose to drop an event immediately and inform the FROB about the dropped event[9], or it may choose to wait to see if required resources would be available.

# 5. EVALUATION

## 5.1 Prototype Implementation

We have implemented the FROB runtime using the Java programming language and tested it on J2ME CLDC[10] [36] version 1.1 from Sun Microsystems. This platform is targeted at network-connected mobile and embedded devices. The platform is for devices typically having a 16- or 32-bit processor running at about 12-32 MHz and having 160 KB ROM and 192 KB of RAM.

Using the prototype implementation of the FROB runtime, we have implemented the scenario as described in Section 3.8 and made it run under J2ME CLDC. To this end we had to augment the standard installation with optional packages from the J2ME Wireless Toolkit version 2.2 to give the platform audio capabilities. For the actual implementation of the mobile multimedia scenario, we are using a subset of the Java language, such as not using loops, as required by our model.

Compared to the scenario in Section 3.8 we have had to divert from it in a few areas, which stems from the fact that the KVM virtual machine included in J2ME CLDC has limitations, which restrict it to dynamically loading classes only from its deployed jar-file and thus not from remote locations. Hence, FROBs can only exchange classes which are known at deployment time. In the J2ME CLDC version of the FROB runtime it is not possible to send an unknown implementation of for instance actions or other executable code entries in the dictionary. In other words, compared to the scenario, the actual sending of code from audio provider to audio client had to be removed. Instead, the audio client and the audio provider only exchange raw data – the audio stream.

Thus, the adaptability of these FROBs is currently limited in that they cannot exchange new functionality through new code; they can only communicate data. Despite the resulting adaptability limitation mentioned earlier, the FROBs can still adapt to changes in resource availability and adjust their service level, as long as these capabilities are implemented by code already available at deployment time.

## 5.2 Base FROB Model in Java

On our current Java platform, a FROB is represented by an abstract class encapsulating a dictionary, which is then subclassed by various implementations – an excerpt of one such subclass is depicted in Figure 15. The corresponding constructors then simply populate their FROB dictionary with their respective interface, triggers, conditions and actions. As such, subclassing is merely used as a *structuring* mechanism, rather than for actual typing purposes.

[9] Such a notification would of course be event based, and would be received by the FROB if the FROB accepts such events according to its interface.

[10] Using the MIDP 2.0 profile

Event types, conditions and actions are all implementations of their respective Java interfaces; Actions, for instance, receive different implementations for handling discovery events, streaming events etc., as needed for the scenario. Likewise, events are represented by a marker interface without any method definitions as the semantics of a given event is irrelevant to the runtime. Triggers are instances of a class representing a tuple containing the name of a condition and an action. Since Java does not directly support predicate expressions in the language, conditions are represented as standard logical expressions encapsulated inside methods.

```
public class AudioStreamingFROB extends FROB {

 /**
 * Constructor.
 */
 public AudioStreamingFROB() {

  // add initialization action
  getDictionary().put("init",
    new AudioStreamingInitAction());

  // add action for handling announcement requests
  getDictionary().put("announcementRequest",
    new AnnouncementRequestAction());

  // add condition for handling announcement requests
  getDictionary().put("announcementRequestCondition",
    new AnnouncementRequestCondition());

  // get trigger list
  final LinkedList tList = Helpers.getTriggerList(this);

  // associate condition and action in trigger
  tList.add(new Trigger("announcementRequest",
    "announcementRequestCondition"));
  ...
 }
}
```

**Figure 15: Excerpt of a Java-based Representation of a FROB**

## 5.3 A Closer Look at The FROB Runtime

The FROB model relies on a common runtime to be first installed on all devices on which deployed FROBs are to be executed. This runtime – whether built on top of a virtual machine or itself being integrated into a specialized virtual machine – ensures independence from the underlying hardware. It is responsible for hosting and executing FROB-based applications as well as monitoring and controlling resources. It also has to provide an infrastructure for allowing the deployed FROBs to communicate via typed events. In addition, it must support distribution of code between FROBs and mobility of FROBs between devices.

The runtime enables event-based communication between FROBs (inside and between devices) in a loosely coupled manner. In addition, a set of FROB libraries is provided, that deployed FROBs can use (via event communication) to, for instance, discover other FROBs on the network, to set up collaboration between FROBs, to manage migration of FROBs, or to receive a notification if a given FROB suddenly becomes unavailable on the network. An illustration of the interaction between the runtime and FROBs is depicted in Figure 16.

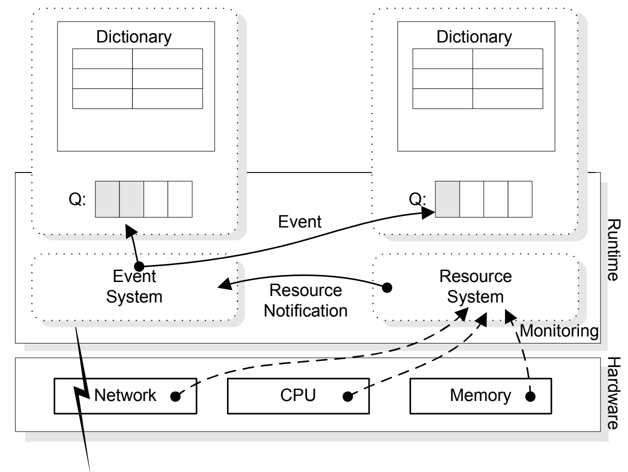



**Figure 16: A platform with two deployed FROBs**

The base infrastructure of the FROB runtime consists of three main components; an event manager, a resource manager and a number of FROB containers.

1. The event manager is responsible for enabling event-based communication between FROBs. The communication between FROB runtimes is based on simple broadcast-based communication. For this event handling, the event manager uses two threads, which are respectively listening for incoming events as well as distributing them naively (i.e. giving a copy to each FROB without looking at FROB interfaces) among the deployed FROB containers.

2. The resource manager is responsible for monitoring and controlling the resources of the runtime. Since neither of the standard Java virtual machines provide the required level of resource control needed for the FROB runtime to monitor and allocate resources, the resource control of the FROB runtime prototype is based on a simple simulation built directly into the runtime.The resource control is simulated by hardcoding a fictitious amount of CPU, memory and bandwidth into the resource manager from which the conditions can request allocations. Upon having enough spare resources to meet the requirements expressed in a condition, the resource manager temporarily subtracts the resources required, after which the action associated with the condition is executed. Once the action has been executed, the allocated resources are handed back to the resource manager.

3. Each FROB is executed within an enclosing, single-threaded container, which holds the associated event queue of the FROB. The container, which is considered part of the runtime, is assigned a thread when

loaded, which (a) matches accepted events against the FROB interface, (b) matches accepted events against conditions, and (c) executes actions. Nothing prevents a simple thread pool enabling sharing of threads between FROB thereby reducing the number of threads and avoiding statically assigning a thread to a FROB.

As a general principle, the FROB runtime ensures that new events can be received continuously while accepted events in the FROBs are being processed. The runtime dedicates at most one thread per deployed FROB for processing of accepted events and execution of actions. Since each FROB only has a single thread, there is no internal synchronization problems stemming from concurrency. This does however not preclude a solution where parallelism, when offered by the underlying OS and hardware, would be actively exploited inside the platform, but this would have to be transparent to the programmer.

## 5.4 Measurements

We performed some measurements on the prototype implementation of the FROB runtime as well as the implementation of the scenario as described in Section 3.8. We focused on the code size and memory usage. The results are depicted in Figure 17.

| | Code Size | Memory Usage |
|---|---|---|
| Runtime | 63 | 13 |
| Libraries | 26 | 1 |
| Application | 28 | 4 |
| Total | 117 | 14 |

**Figure 17: Empirical Measurements Code Size (uncompressed class files) and Memory Usage of FROB Elements (in kilobytes)**

Compared to the typical memory budget of a KVM deployment of respectively 256 kilobytes [39], the FROB runtime in its prototype implementation thus entails a memory overhead of approximately 5%.

## 5.5 Virtual Machine Support

Based on the experiences gained with the implementations of our prototype, combined with an investigation of other virtual machines targetted for resource constrained mobile devices, among others the SmallTalk based OSVM [34] from OOVM, we have constructed a list of required features that a virtual machine designed for the FROB platform should provide in order to support all features of an ideal FROB platform.

- Resource Control – low-level control of resources such as CPU, memory, bandwidth etc. to enable allocation of resources and provide guarantees for resource availability to meet resource requirements expressed in the conditions of a FROB.

- Dynamic Classloading – for code distribution purposes, loading of classes sent between FROBs, thereby enabling exchange of new code such as actions.

- Class Unloading – for memory consumption purposes, unloading of classes no longer needed by the FROB runtime.

- Broadcasting – to enable the FROBs to discover and communicate.

Though there seem to be some way to achieve dynamically loading and unloading of classes in some virtual machines for non-mobile devices[11], these capabilities – or parts of them – seem typically to be left out of platforms for small devices such as J2ME CLDC and OSVM.

## 5.6 Overall Evaluation

On the basis of the above prototype, we were able to demonstrate a convincing example of non-trivial mobile computing, to present some encouraging concrete numbers and to collect detailed information on the issues in implementing the proposed computing model.

More fundamentally, the ingredients of our model (time-slicing, event-based, dynamic change of interfaces, code mobility) were selected deliberately and carefully in response to the limitations of standard programming conventions.

Further evaluation will consist in weighing up the consequent strengths and weaknesses of the FROB model as an answer to the challenge of implementing agile, light-weight adaptive resource-aware systems. One of the questions that will have to be addressed is whether the model is appropriate in terms of programmer productivity: does it provide the right abstractions for mastering the complexity, does it contribute to reduce errors when implementing highly connected, but resource-frugal software? It is true that several design choices in the model, such as preventing explicit loops, seem to lead to a rather low-level programming style, but we believe that proper language constructs can at least partly compensate for that. Another difficulty stems from the new level of adaptability that FROBs offer by enabling code to be dynamically modified, at a very fine granularity. This facility may become a trap if used without proper methodology. We are currently developing libraries for FROB-based programming. In particular, we are abstracting some of the main FROBs in our streaming scenario in the form of a reusable library, along the line of [8]. More generally, the idea is to end up with fine-grained libraries of event types and triggers. Particularly, we are working on such libraries for improving resource awareness capabilities as used in the scenario.

Another question is whether the model leads to efficient implementations, with lower system load, and does it favour good throughput of event handling? Whereas the current prototype was built in Java on top of a KVM, a dedicated language and VM will be more appropriate to reduce discrepancies between the model and its implementation, and hence provide fair numbers. We are therefore experimenting with a specialized virtual machine which integrates the functionality of the FROB runtime. This shall also enable

[11] For instance, the Java J2SE virtual machine can achieve these capabilities through the use of custom classloaders.

more direct control over physical resources than what is possible with virtual machines such as KVM and OSVM. Many behind-the-scene optimizations are furthermore to be explored, such as to route internal, self-addressed events more directly than in the general case.

# 6. CONCLUDING REMARKS

This paper discusses the challenge of devising a computing model for mobile devices. The paper answers some of the questions underlying that challenge in the form of the FROB computing model and runtime.[12]

On the basis of a prototype running on the Java platform for resource-constrained devices, we were able to demonstrate a demanding example of peer-to-peer mobile computing, to present some encouraging concrete deployment figures and to collect detailed information on the issues in implementing the proposed computing model.

Further experiments are needed and many FROB aspects need to be refined. Among these aspects, abstractions for code reuse have not been discussed and further research is needed to explore how (abstract) classes, (open) interfaces and inheritance could be appropriately used in our context [27].

## Acknowledgements

This work is conducted under the PalCom project, financed by the European Community under the Future and Emerging Technologies initiative. We are very grateful to our partners in the project for many interesting discussions, in particular Peter Andersen, Lars Bak, Erik Ernst, Monique Calisti, Steffen Grarup, Dominic Greenwood, Kasper V. Lund, Boris Magnusson, Martin Odersky, and Reiner Schmidt.

# 7. REFERENCES

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, 1986.

[2] G. Agha and C. J. Callsen. ActorSpace: an open distributed programming paradigm. In *Proceedings of the 4th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 23–32. ACM Press, 1993.

[3] J. Armstrong. The development of Erlang. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 196–203. ACM Press, 1997.

[4] K. Arnold, B. O'Sullivan, R. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification.* Addison-Wesley, 1999.

[5] G. Back, W. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI'2000)*, San Diego, CA, USA, oct 2000.

[12] We have chosen a name (FROB) for our computing model that hopefully conveys its experimental nature, rather than names of dead mathematicians like Pascal, Occam, Euclid or Erlang.

[6] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Comput. Surv.*, 21(3):261–322, 1989.

[7] W. Binder and J. Hulaas. A portable CPU-management framework for Java. *IEEE Internet Computing*, 8(5):74–83, Sep./Oct. 2004.

[8] A. P. Black, J. Huang, R. Koster, J. Walpole, and C. Pu. Infopipes: an abstraction for multimedia streaming, multimedia systems. *Multimedia Middleware*, 8(5):406–419, 2002.

[9] F. Boussinot and R. de Simone. The Esterel language. In *Proceedings of IEEE*, volume 79, pages 1270–1282, 1991.

[10] E. Cheong, J. Liebman, J. Liu, and F. Zhao. TinyGALS: a programming model for event-driven embedded systems. In *Proceedings of the ACM symposium on Applied computing*, pages 698–704. ACM Press, 2003.

[11] G. Czajkowski and L. Daynès. Multitasking without compromise: A virtual machine evolution. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, Tampa Bay, Florida, oct 2001.

[12] G. Czajkowski, S. Hahn, G. Skinner, P. Soper, and C. Bryce. A resource management interface for the Java platform. *Software Practice and Experience*, 35(2):123–157, nov 2004.

[13] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[14] A. Dunkels, B. Grnvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors 2004*, 2004.

[15] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[16] P. T. Eugster, R. Guerraoui, and C. H. Damm. On objects and events. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 254–269. ACM Press, 2001.

[17] A. Frei, A. Popovici, and G. Alonso. Event based systems as adaptive middleware platforms. In *Proceedings of the 17th European Conference for Object-Oriented Programming*, 2003.

[18] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[19] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition.* Addison-Wesley, 2000.

[20] C. E. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3), June 1977.

[21] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th international conference on Architectural support for programming languages and operating systems*, pages 93–104. ACM Press, 2000.

[22] J. Hulaas and W. Binder. Program transformations for portable CPU accounting and control in Java. In *Proceedings of PEPM'04 (2004 ACM SIGPLAN Symposium on Partial Evaluation & Program Manipulation)*, pages 169–177, Verona, Italy, August 24–25 2004.

[23] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, 1988.

[24] W. A. Kornfeld and C. E. Hewitt. The scientific community metaphor. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(1):24–33, January 1981.

[25] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[26] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN conference on Programming Language design and Implementation*, pages 260–267, 1988.

[27] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. pages 107–150. Research directions in concurrent object-oriented programming, MIT Press, 1993.

[28] Microsoft. Microsoft .NET framework. http://www.microsoft.com/net.

[29] Microsoft. *DCOM Technical Overview (Microsoft White Paper)*, 1999.

[30] A. Mukhija and M. Glinz. A framework for dynamically adaptive applications in a self-organized mobile network environment. In *ICDCS Workshops*, pages 368–374, 2004.

[31] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM symposium on Operating systems principles*, pages 276–287. ACM Press, 1997.

[32] OMG. *The Common Object Request Broker: Architecture and Specification*, February 1998.

[33] K. Raatikainen. A new look at mobile computing. In *Proceedings of Academic Network for Wireless Internet Research in Europe (ANWIRE) Workshop in Athens*, May 2004.

[34] Resilient. *OSVM*. http://www.oovm.com.

[35] J. Robert H. Halstead. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[36] Sun Microsystems. *Connected Limited Device Configuration*. http://java.sun.com/products/cldc.

[37] Sun Microsystems. *Mobile Information Device Profile*. http://java.sun.com/products/midp.

[38] Sun Microsystems. *Java Remote Method Invocation – Distributed Computing for Java (Sun Microsystems White Paper)*, 1999.

[39] Sun Microsystems. *J2ME Building Blocks for Mobile Devices*, 2000. http://java.sun.com/products/cldc/wp/KVMwp.pdf.

[40] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, R. Jeffers, T. S. Mitrovich, B. R. Pouliot, and D. S. Smith. NOMADS: toward a strong and safe mobile agent system. In *Proceedings of the 4th International Conference on Autonomous Agents (AGENTS-00)*, NY, jun 2000.

[41] UPnP Forum. *Universal Plug and Play Specification v. 1.01*. http://www.upnp.org.

[42] M. Welsh and D. Culler. Overload management as a fundamental service design primitive. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, 2002.

[43] A. Yonezawa and M. Tokoro. *Object-oriented concurrent programming*. MIT Press, 1987.