Frugal Mobile Objects

Benoît Garbinato[2], Rachid Guerraoui[1], Jarle Hulaas[1],
Maxime Monod[1], and Jesper H. Spring[1]

[1] Ecole Polytechnique Fédérale de Lausanne (EPFL),
School of Computer & Communication Sciences,
CH-1015 Lausanne
[2] Université de Lausanne,
Ecole des HEC,
CH-1015 Lausanne

Abstract. This paper presents a computing model for resource-limited mobile devices. The originality of the model lies in the integration of a *strongly-typed event-based* communication paradigm with abstractions for *frugal* control, assuming a small footprint runtime.
With our model, an application consists of a set of distributed reactive objects, called FROBs[3], that communicate through typed events and dynamically adapt their behavior reacting to notifications typically based on resource availability. FROBs have a logical time-slicing execution pattern that helps monitor resource consuming tasks and determine resource profiles in terms of CPU, memory, and bandwidth. The behavior of a FROB is represented by a set of stateless first-class citizens together with an indirection to the state and actual behavioral references of the FROB. This facilitates the dynamic changes of the set of event types a FROB can accept, say based on the available resources, without requiring a significant footprint of the underlying FROB runtime.
We demonstrate the usability of the FROB model through our J2SE-based prototype and a peer-to-peer audio streaming scenario where an audio provider dynamically adjusts its quality of service by adapting to demand. The performance results of our prototype convey the small footprint of our FROB runtime (86 kilobytes). We also augmented the KVM to enable resource profiling with however a negligible overhead (less than 0.5%) and a decrease in speed of the virtual machine of at most 7%.

It is not the strongest of the species that survives, nor the most intelligent, but the one most responsive to change. C. Darwin

1 Introduction

Motivation

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

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

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, sometimes 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 limited in terms of processing power, storage and bandwidth, for these may or not be available, depending on the mobility pattern and the solicitations. Software components running on such devices are typically supposed to automatically discover each other on the network and join to form ad-hoc peer-to-peer communities enabling mutual sharing of each others functionalities by offering and lending services. Underlying communication substrates might include wireless LANs, satellite links, cellular networks, or short-range radio links.

In an ever changing environment of resource-constrained devices, the *frugality* of software components is paramount to their operation. Besides conveying that these components are simply "small" (the meaning of which depends on the underlying technologies), frugality also conveys notions of resource awareness and adaptivity. More specifically, this implies being aware of resources consumed by the software, dynamically adjusting the quality of service following changes to the environment, and making sure that resources are in fact available when certain tasks are launched.

To illustrate this, imagine a mobile service offering to stream some data, say 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.

More generally, we believe that three principles should drive the design of a computing model for resource-constrained mobile devices:

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 dynamic and mobile environments. As discussed above, the software component of a device should adapt to its changing environment and it 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 [30] 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 be 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 available resources. This calls for a computing model where the components are made aware of the resources they use. 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 (RPC)* abstraction (e.g., [27, 29, 35]). The RPC abstraction aims at promoting the porting of centralized programs in a distributed context. Clearly, RPC makes little sense when the invoker does not know the invokee, or does not even know whether there is one at a given point in time. Some of the extensions to the RPC abstraction, including *futures* [31] (also called *promises* [23]) 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 resource-constrained devices, is rather challenging. The aim of our work is precisely to face that challenge and this paper is a preliminary step in that direction.

Contributions

This paper presents a computing model based on frugal objects, called *FROBs*, which are supposed to be deployed and executed on a small memory footprint runtime running on a resource constrained device.

- Computing is triggered by *typed events* that regulate the possibly anonymous and asynchronous communication between FROBs ((1) in Figure 1). A FROB can specify the type of events it can process, and how, through *behavioral objects* ((3) in Figure 1). Each such object is a sequential unit of computing, bound to the handling of a single event type, and representing a partial behavior of a FROB. At any point in time, the set of behavioral objects in a FROB complies with its external *interface*, i.e., the set of event types it is capable of handling ((2) in Figure 1). Upon receiving an event, the runtime matches it against the interface to determine whether to accept the event for further processing or reject it.
- Besides preventing casting errors and acting as a filtering mechanism [15], our typed event model makes it possible to adopt a fine-grained serialization scheme that exploits (1) the decentralized representation of a behavior, and (2) its binding to event types. Our serialization mechanism does thus not rely on a general reflective scheme and is memory efficient. Since the serialization capabilities are bound to, and integrated with the behavioral objects, these act as fully functional units of distribution, which can be exchanged between FROBs.
- Key to supporting adaptivity with minimal underlying footprint, is the stateless representation of a FROB behavior as a set of first-class citizens within the FROB, together with a level of indirection to its state and behavioral references. This enables easy replacement of the FROB behavior during execution.

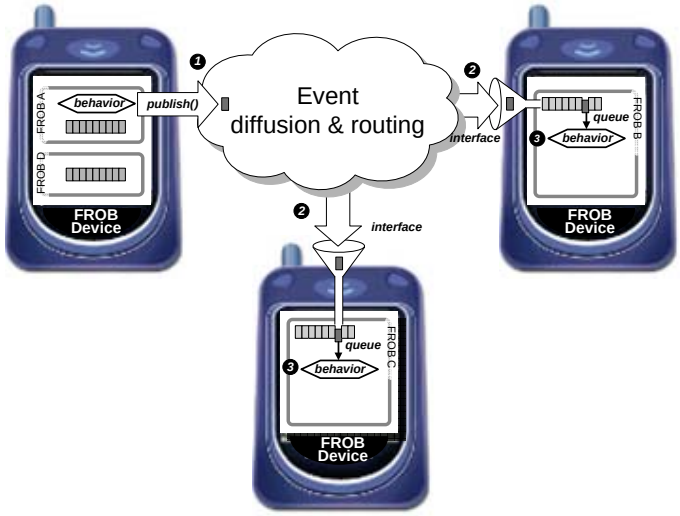


Fig. 1. Event-based interacting FROBs

- FROBs are inherently threadless and one behavioral object is executed at a time. Long running procedures are split up into small, short-lived event-based behavioral objects. The resource requirements of these individual behavioral objects are thus limited in terms of actual resource amount needed and required duration.
- The FROB runtime continuously monitors availability of internal resources on the device (CPU, memory, bandwidth, etc.) and deduces resource profiles when executing behavioral objects. Upon detecting a significant change in resource availability, or if prevented from executing a behavioral object based on its requirements, expressed as a resource profile, the runtime informs the FROBs deployed on the device about the change. These notifications are themselves provided as regular typed events that the FROBs can choose to react to by adjusting their external interfaces.

In retrospect, and to summarize, our contribution is not a single revolutionary abstraction, but rather a way to integrate a typed event-based communication paradigm with a resource-aware adaptive control scheme. The challenge is precisely in achieving that integration under the constraint of an underlying runtime with a small footprint.

Evaluation

We implemented a prototype of our computing model on top of the J2SE [34] platform (version 1.4.2_07). We considered this platform instead of the J2ME CLDC [33], which is usually targeted at resource-constrained devices, because

J2SE includes a classloader framework enabling unloading of classes[4] and a richer set of networking capabilities whereas its J2ME counterpart only comes with very basic abstractions for point-to-point communication.

We experimented our model and prototype through an ad-hoc (peer-to-peer) audio streaming scenario between mobile devices, which turns out to be challenging in terms of resource-awareness and adaptivity. To conduct performance tests, we used a machine fitted with an AMD Athlon 3200+ processor, 2 GB of RAM, 100 Mbps network card and under Debian linux (kernel 2.6.8-2-k7). This machine hosted both the audio provider and the clients used in the evaluation.

Our performance measures conveyed (1) a small footprint of our FROB runtime (86 kilobytes), (2) a memory overhead of a factor 8 for our fine-grained way of representing behavior (with respect to an object-oriented representation as in Java), and (2) an insignificant performance drop due to our added level of indirection (0.5% in our scenario).

To experiment the generation of resource profiles, we also modified the KVM of Java J2ME CLDC [34] and conducted some performance evaluations to determine the cost of the profile generation. These modifications changed the size of the KVM by a negligible amount (less than 0.5%) and decreased its overall speed by at most 7%.

Roadmap

Section 2 describes the FROB computing model. Section 3 illustrates the main concepts through the audio streaming scenario. Section 4 presents our performance measures. Section 5 positions the FROB computing model with respect to alternative approaches. Section 6 gathers some final remarks.

2 The FROB Computing Model

2.1 A Static View of a FROB

A FROB conceptually consists of (Figure 2): (1) an external interface (deduced from the set of behavioral objects), (2) a FIFO-ordered queue of received events, (3) a set of fine-grained behavioral objects to manipulate the state of the FROB, and (4) the actual state representation.

Both the state and the behavioral objects of a FROB are contained in named slots in a data structure within each FROB called a *dictionary* (see (5) in Figure 2). The notion of dictionary is similar to that of *slots* in the Self language [32]; a slot can contain either state or code.

[4] The classloader framework is needed to avoid a constant increase in heap size, which over time follows from the frequent replacement of behavioral objects (and thus loading of classes, allocated in the heap) since Java per default (and in particular J2ME CLDC) cannot unload classes (only their instances). The choice of Java itself does not imply that the runtime or our mechanisms are bound to Java nor to any other particular language or execution platform. Java essentially provides a convenient implementation basis for deploying software throughout heterogeneous networks.

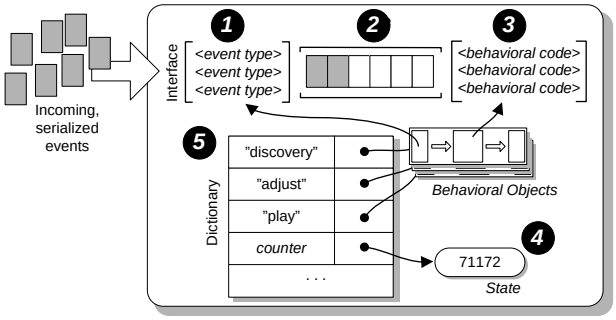


Fig. 2. Conceptual view of a FROB

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

When receiving events, the runtime places incoming events into the event queue of a FROB if they match one of the event types in its external interface. When events are present in the queue of a FROB, the runtime looks up (in the dictionary) and executes the behavioral object capable of handling the typed event.

FROBs are encapsulated entities that do not share state (i.e., entries in the dictionaries) – the behavioral objects always run isolated from each other. This combination eliminates the need for synchronization on entries in the dictionary.

2.2 Typed Events

Events are the basic entity to which FROBs react, i.e., an event might cause a behavioral object in a FROB to be executed. They serve as communication units between multiple FROBs, whether deployed on different devices or on the same one.

Events are typically *published* by the FROBs, or possibly by the runtime itself following some internal event, and distributed between the devices using the communication infrastructure provided by the FROB runtime. An event is accepted by a FROB only if the latter has *subscribed* to the type of that event, i.e., if the FROB has that event type in its interface. Unlike in many statically typed systems, FROBs have dynamic types as their capabilities may change throughout their lifetimes.

FROBs hence communicate through a topic-based publish-subscribe interaction paradigm, where the topic is the type. This event-based scheme is, resource-wise, a cheap alternative to multi-threading systems that are considered expen-

sive in terms of stack management and over-provisioning of stacks, as well as locking mechanisms [13].

Although a publish-subscribe scheme is inherently anonymous and asynchronous, it does not preclude coupled forms of interactions. One could easily encode a point-to-point interaction scheme by having the identifiers of the interacting FROBs as parts of their event type. The actual dissemination of events in a distributed and mobile environment following a publish-subscribe pattern is out of the scope of this paper and is a research subject on its own [6].

2.3 Fine-grained Serialization

In order to collaborate, the FROBs first have to discover each other and then initiate collaboration. FROBs collaborate by exchanging events and by – as part of the collaboration initiation – exchanging the necessary behavior to interpret the events, i.e., the FROBs adapt to each other to collaborate. This exchange of behavior is required as the particular capabilities needed to interpret the events being sent might not be present on the FROB receiving the events. To perform this exchange of behavior and data over the network, a *serialization mechanism* is required.

In contrast to a resource consuming, general-purpose serialization mechanism typically found in traditional distributed runtimes, as mentioned before, we consider a *fine-grained* mechanism where each behavioral object is required to provide its own (de-)serialization capabilities. As such, each behavioral object contains the functionalities to deserialize the incoming event type that it handles and serialize any typed event that it publishes as illustrated in Figure 3.

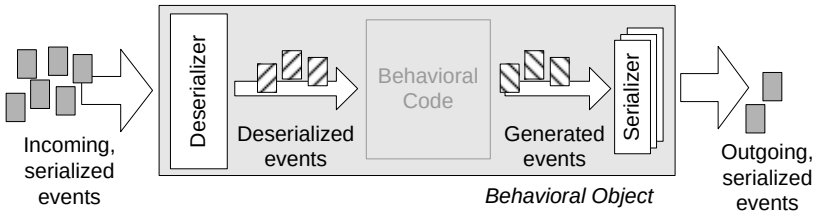


Fig. 3. Behavioral object with deserializer and serializers

We exploit the very fact that communication between FROBs occurs through the typed events. As mentioned earlier, each behavioral object is bound to and its execution is triggered by a particular typed event. The execution of the behavioral object may, however, cause a number of new typed events to be generated and published to other FROBs, or even to the FROB itself. In other words, the (de-)serialization capabilities required for each behavioral object are limited, static and all known at compile-time.

By bundling the actual (de-)serialization capabilities with the behavioral objects using them, the specific capabilities, so to say, follow their *user*, and thus make up a single, fully functional distribution and deployment unit. With these units, it is possible to have only the minimal (de-)serialization capabilities loaded by the runtime. Once some behavior is no longer needed, and thus gets unloaded by the runtime, its (de-)serialization capabilities get unloaded too. Thus, the coupling between the fine-grained behavioral representation and the *fine-grained serialization* mechanism is a memory-efficient combination suited for resource-constrained devices.

Conceptually, each behavioral object provides its own (de-)serialization capabilities, a fact which results in a potential memory overhead in situations where the same capabilities are needed in multiple behavioral objects on the same device. The runtime can circumvent this potential overhead by transparently sharing these capabilities between behavioral objects based on the event type, and thereby only loading a single instance of the functionality.

2.4 Indirectional Reflection

As opposed to a general-purpose class-based reflection scheme, we rather adopt an *indirectional reflection* based on a fine-grained representation of every FROB in the form of a state representation, together with a set of first-class citizens: behavioral objects. This fine-grained granularity allows for flexible modification of the FROB. Through the separation of state and behavior within the FROB, the behavioral objects are immutable, which at the same time makes them suitable units of replacement as no state is lost during the replacement.

Each behavioral object has access to the dictionary of the FROB to which it belongs, and can manipulate it through appropriate primitives (for looking up, adding and removing entries) during its execution.

The name/value pairs in the dictionary provide a *level of indirection* which is key to our reflection capabilities. Using this level of indirection, all references to state and behavioral objects go through these name/value pairs, which thus enables the actual values to be easily replaced without replacing the references as illustrated in Figure 4. In fact, this also enables behavioral objects to cause their own replacement.

Roughly speaking, a FROB adapts by changing behavior, i.e., what capabilities it can provide, or how it provides them. This behavioral change materializes in (1) keeping the current set of behavioral objects contained in the dictionary, but making adjustments to it, or (2) by actually extending, reducing or modifying the behavioral objects within that set.

2.5 Logical Time-Slicing

FROBs are inherently threadless. Instead, threads are assigned to the execution of FROBs (or rather their behavioral objects) by the runtime in a time-slicing scheme. In this scheme, an event in some FROB's queue represents a request

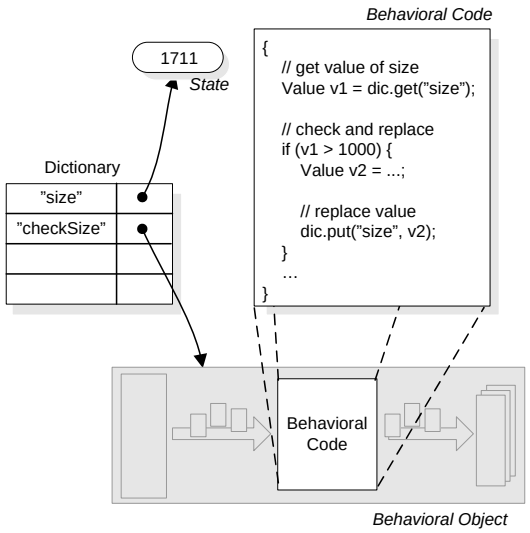


Fig. 4. Dictionary with state and behavioral objects

for some time-slice, which is granted when the behavioral object consuming that event is executed.

The FROB runtime does not dictate a specific threading model for executing the behavioral objects. It ensures, however, that (1) a behavioral object, for which a typed event matching the interface of the FROB has been received, will eventually be executed on the event, and (2) no two behavioral objects of the same FROB can execute concurrently. Combined with the time-slicing scheme, this gives the runtime explicit control points between executions. Besides concurrency control and resource-profiling motivations, these explicit control points make it easier to manipulate (i.e., to perform behavioral changes) the FROB and even leaves the possibility to checkpoint or migrate it. Specifically, since at any explicit control points no thread is active within the FROB, its state is well-defined and it can thus easily be captured or manipulated.

Once executed by the runtime, behavioral objects are allowed to run to completion. The resource requirements of these individual behavioral objects are thus limited in terms of actual resource amount needed and required duration. These requirements are associated to each behavioral object expressed in a resource profile used by the runtime. (We will come back to resource profiling later.) This scheme of small, short-lived execution units is also promoted by the fact that the FROB programming model precludes the use of recursive calls, forks, and synchronization primitives in the behavioral objects. In particular, this prevents the execution of a behavioral object from thread monopolizing the CPU. Instead, the behavioral objects systematically yield the control to the runtime. In

addition, since the computing model defines no blocking primitives, a FROB has no way to compromise liveness.

2.6 Resource-Profiling

As illustrated in Figure 5, the FROB runtime constantly monitors the availability of internal resources such as CPU, memory, bandwidth etc. Upon detecting significant changes to resource availability, according to some predefined threshold values, the runtime publishes notifications enabling FROBs to possibly react and change behavior.

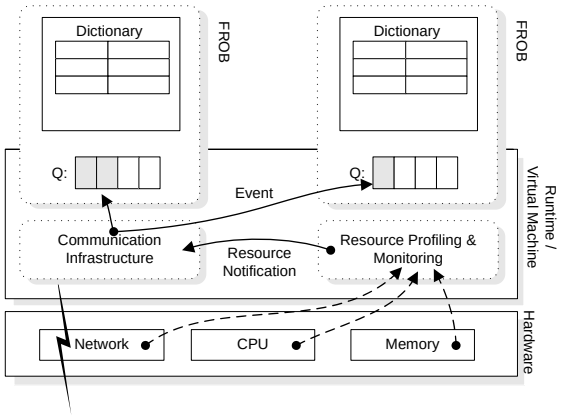


Fig. 5. Resource profiling and monitoring in the FROB runtime

Attached to each behavioral object is a *resource profile* which describes the amount of resources (CPU, memory, bandwidth, etc) the object required during its execution. These profiles are generated by the FROB runtime by measuring the actual execution and are attached to the behavioral object subsequently. Through these resource profiles, the runtime has a prediction of the resource requirement for a future execution. Throughout the lifetime of a behavioral object, its execution pattern might change, e.g., by executing differently (and thus have different resource requirements) depending on the actual event received. To try to limit the distorted effects that such execution variations have on the prediction, the runtime tries to compensate by keeping track of certain historical executions, and thus the profile gets more and more accurate the more the behavioral object gets executed.

As part of its event scheduling strategy, the runtime uses the resource profile associated with each behavioral object to evaluate the ability, at a given point in time, to execute the behavioral object based on the resource requirement stated in the profile compared to the resource availability on the device. By comparing

the two, the runtime can determine if there are enough resources to execute a behavioral object.[5] If this is not the case, the execution of the behavioral object is postponed and an event is published to the FROBs deployed on the runtime, notifying them about the current resource shortage. Upon receiving such an event, the FROBs can then try to collaborate by freeing up resources, i.e., by adapting.

If a FROB desires to adapt to such a notification by actually replacing behavioral objects, the resource profiles can be used by the FROB as an indicator for finding an alternate behavioral object that uses less resources, or uses resources differently, e.g., more bandwidth, but less CPU and/or memory, such that the resource shortage can be lifted.

For instance, if the resource availability is reduced within a device, a FROB might adapt using strategies that try to either reduce the current resource consumption or tries to find alternative sources of resources. Several strategies are possible.

1. *Unload Behavior* – The FROB can try to unload unused or infrequently used behavioral objects present in the dictionary, in order to try to free resources. Unloading behavioral objects might have a limited effect on memory, though, as the behavioral objects themselves are stateless and thus do not carry a lot of data;
2. *Adjust Quality of Service* – The FROB can try to offer the same service at a lower quality in such a way that its resource consumption better fits with the newly announced resource availability. Specifically, this adjustment is done by adjusting or replacing behavioral objects using the resource profiles attached to the behavioral objects to determine which fit better to the resource availability;
3. *Migrate* – The FROB can try to migrate from one device to another following resource availability changes that motivates the execution to be continued on another device, e.g., the presence of a more resource-rich computing environment, or the reception of a notification that the computing environment on which the FROB is running is about to close down, e.g., due to power exhaustion.

3 Illustrating FROBs

To demonstrate the viability of our FROB model, we designed and implemented an audio streaming scenario involving mobile devices. The scenario does not rely on any centralized server and poses several challenges in terms of frugality. As such, the scenario includes resource adaptivity following changes in connectivity, e.g., when a service disappears and exposes problems inherent to resource aware devices/runtimes, like, for instance, how the audio provider should react as the

[5] The FROB runtime uses best-effort to predict if enough resources are available to execute a behavioral object. As such, there is no guarantee that the behavioral object can run to completion without experiencing resource-related errors.

number of requesting audio clients is growing, how to preserve the same stream, when to degrade, etc.

Though the audio streaming scenario is implemented in J2SE, as mentioned earlier, and the code excerpts presented in the following thus are exemplified using the Java programming language, as we pointed out, the FROB model is not specific to any specific programming language or platform.

3.1 The Audio Streaming Scenario (The Network of FROBs)

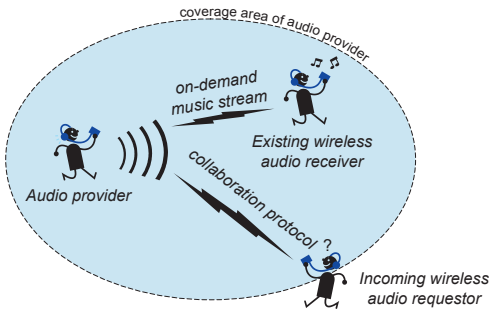


Fig. 6. The audio streaming scenario

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 6.

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 audio provider and client initiates each a discovery protocol.

Having discovered each other, the FROBs start negotiation for collaboration, i.e., the client FROB requests the audio providing FROB for permission to use its service – receive the audio stream. Having gained permission, the client receives the behavioral objects to decode the audio stream after which the audio streaming starts.

3.2 Request for Collaboration (Behavioral Objects)

Programming in the FROB model consists in programming a set of behavioral objects, which are present in named entries in the dictionary of a FROB. When a behavioral object is executed (following the occurrence of an event), it will typically manipulate entries in the dictionary (that represents other behavioral objects or state), and generate other typed events which are then published to other FROBs. This is illustrated in Figure 7, which depicts an excerpt of a

behavioral object from our Java-based implementation responding to requests for collaboration.

Specifically, Figure 7 shows how the behavioral object uses entries in the dictionary (line 4 and 9) to store and lookup ongoing collaborations.

```
 1: public void execute(Runtime rt, Dictionary dic, CollaborationRequestEvent event)
        throws FROBException {
 2:   ...
 3:   // check if collaboration with client already exists
 4:   if (dic.get(event.getClientId()) != null) {
 5:     // cancel collaboration request; collaboration exists
 6:     ...
 7:   }
 8:   // create collaboration state
 9:   dic.put(event.getClientId(), new CollaborationState());
10:   ...
11: }
```

Fig. 7. Excerpt of behavioral object; managing ongoing collaborations through a dictionary

As conveyed by Figure 7, the code of any behavioral object is accessible to the FROB runtime through a method called `execute`. Upon receiving the correctly typed event, this method is invoked by the FROB runtime along with references to (1) the runtime on which the FROB is deployed (parameter `rt`), (2) the dictionary of the FROB (parameter `dic`), and (3) the event that caused the execution of the behavioral object (parameter `event`).

3.3 Responding to a Collaboration Request (Typed Events)

In our Java-based implementation, each event type must implement a marker Java-interface `Event`, which acts as our base event type. Through this interface, we provide a set of typed events (and behavioral objects for handling these events) as part of the standard protocols used by the FROBs, e.g., as a collaboration and negotiation protocol.

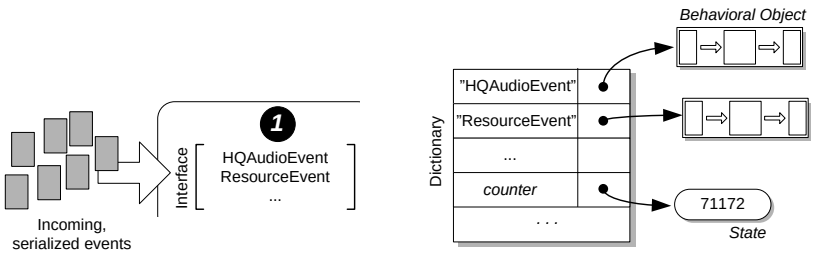
To publish a (typed) event, a behavioral object is expected to serialize the event itself using its own fine-grained serialization mechanism. Figure 8 illustrates how to achieve this in our Java-based implementation. To publish typed events, the behavioral object uses the primitive `publish`, which is accessible to the behavioral object through the FROB runtime reference (the parameter `rt` as seen in Figure 7). Using this primitive, Figure 8 shows how the behavioral object publishes (line 7) an event that it serialized using its fine-grained serialization mechanism (line 4 and 5).

As conveyed by Figure 8, when publishing the serialized typed event, we provide an additional parameter – an indication of the event type being published

```
1: // create typed event for responding to request
2: CollaborationResponseEvent crpe = new CollaborationResponseEvent( ... );
3: // serialize response event
4: CollaborationResponseSerializer ser = new CollaborationResponseSerializer();
5: byte[] ba = ser.serialize(crpe);
6: // publish response event
7: rt.publish(ba, new EventType(CollaborationResponseEvent.class));
```

Fig. 8. Excerpt of behavioral object; responding to collaboration request

(the `EventType`) – which is used by the communication infrastructure to match incoming serialized events against the external interface of the FROBs.



(a) Event types in external interface

(b) Corresponding behavioral objects in dictionary

Fig. 9. Representing the external interface using behavioral objects from the dictionary (excerpt of Figure 2)

At any point in time, the FROB runtime uses the set of behavioral objects in the dictionary of the FROBs to create an external interface, which is mapped into subscriptions for event types that the behavioral objects are capable of handling. This is illustrated in Figure 9, which shows how the event types defined in the external interface (Figure 9(a)) corresponds to actual behavioral objects present in the dictionary (Figure 9(b)).

3.4 Downgrading the Audio Stream Quality (Replacing Behavioral Objects)

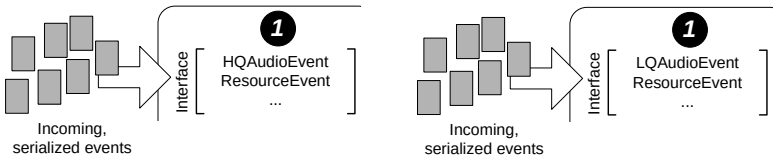
FROBs adapt by changing the set of behavioral objects that they contain in the dictionary, upon receiving a given typed event representing some change.

Figure 10 illustrates how changing behavioral objects is used to downgrade the quality of the audio stream provided.

```
1: // incoming event
2: ResourceEvent re = ...;
3: // downgrade QoS if memory constrained
4: if (re.getMemoryLevel() > 0.9) {
5:   // remove current audio streaming behavior
6:   dic.remove("audioStreamer");
7:   // instantiate new audio streaming behavior and install
8:   dic.put("audioStreamer", new LQAudioStreamingBehavior());
9:   // set buffer size in dictionary
10:  dic.put("bufferSize", new Integer(LQ_BUF_SIZE));
11: }
```

Fig. 10. Excerpt of behavioral object; downgrading a behavior following notification

Changes made to the set of behavioral objects of a FROB is reflected immediately in its dynamic interface such that its subscriptions for typed events match its capabilities in terms of behavioral objects. The effect on the dynamical interface following the behavioral change (seen in Figure 10) is illustrated in Figure 11.



(a) Before behavioral change (b) After behavioral change

Fig. 11. Behavioral change reflected in external interface of FROB - Excerpt of Figure 2

3.5 The Audio Quality of Service (Putting Frugality to Work)

To evaluate the feasibility of the overall scenario, we conducted some measurements of the memory consumption of our audio client and audio provider FROBs when collaborating (both running on J2SE).

In the example, a voluntarily very simple and deterministic resource management policy is used, with the intention to focus on and illustrate the ability of

FROBs to adapt asynchronously and in a fine-grained manner, thereby showing a level of adaption normally not available in other programming models.

Audio Provider. We measured the memory consumption of the audio provider and its ability to adapt its quality of service when put under an increasing workload, i.e., clients to feed streams of audio. Upon receiving a client connection, the audio provider allocates a buffer into which it reads a chunk of audio data that it feeds to the client.

For the sake of the example, we simulated resource shortage notifications on every third connection of a peer, such that the audio provider could react accordingly, by gradually downgrading its quality of service.

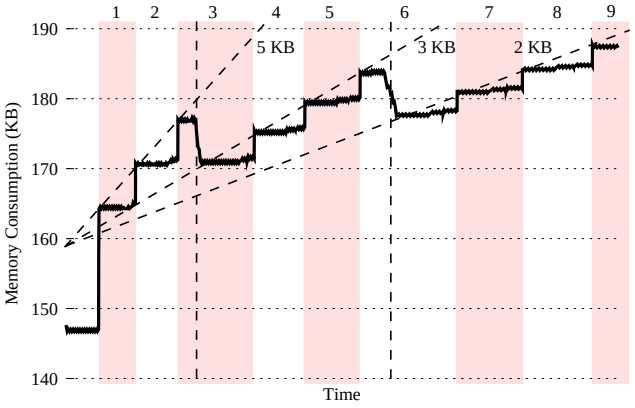


Fig. 12. Memory consumption of audio provider over time with increasing number of audio clients (1-9) and flexible quality of service

The audio provider has been programmed to initially allocate a buffer of size 5 kilobytes per connecting client for the audio data. For the first resource event it receives, the audio provider reduces the sizes of the buffers to 3 kilobytes for every existing client connection, and when receiving the second resource events, it reduces the buffer sizes to 2 kilobytes. The memory consumption from performing this test is depicted in Figure 12, which shows how the audio provider uses resources as nine clients connect with a time interval of approximately 30 seconds between each connection. It can be seen that when the audio provider is idle, the memory consumption is 147 kilobytes, of which the audio server itself is accountable for 8 kilobytes.

Figure 12 shows how each connecting client (marked with shade/non-shade) puts additional resource constraints on the audio provider, but also how the audio provider adapts its quality of service and through that reduces its memory consumption (after the garbage collection by the virtual machine) following the connection of client number three and six.

Figure 12 also shows, by the decreasing angle of the stippled lines, how the relative cost of adding additional audio clients is reduced every time the audio provider adapts following a notification from the resource oracle. This of course is due to the reduction in the buffer size allocated for each connected client following the adaptation by the audio provider.

Audio Client. We measured the memory consumption of the audio client during the playback of the audio stream as well as the change in memory consumption following the (1) loading of behavioral objects needed to receive and play the audio stream, and (2) unloading of the behavioral objects as collaboration ends. In other words, we measured the memory consumption following the fine-grained adaptivity of behavior. The measurements are seen in Figure 13.

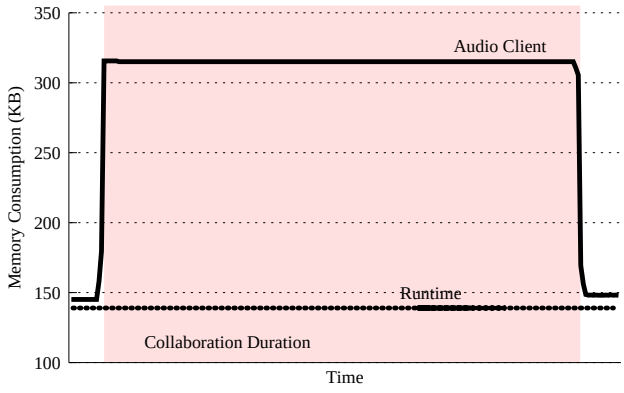


Fig. 13. Memory consumption of audio client over time following loading/unloading of behavioral objects during collaboration

Figure 13 shows how the memory consumption of the audio client increases significantly during the duration of the collaboration with the audio provider (shaded area). Before commencing collaboration, the memory consumption of the audio client is 145 kilobytes, of which the client itself uses of 6 kilobytes. The memory increase is caused by the fact that the audio client receives behavioral objects necessary for retrieving and playing the audio stream from the audio provider. As such, these behavioral objects make use of the Java sound system, which is thus loaded during the collaboration. After having played the stream and collaboration terminates, the audio client unloads all the partial behavior (and the sound capabilities used), and memory consumption goes back to the default level, as also indicated in Figure 13.

4 Overheads

In the following, we evaluate the overhead of the key mechanisms underlying our model (both implemented in J2SE): *indirectional reflection* and *fine-grained serialization*. Then we convey some results from implementing resource profiling in the virtual machine of J2ME CLDC (KVM).

4.1 Runtime Footprint Overhead

The runtime in which we experimented our mechanisms consists of 61 classes taking up 125 kilobytes of memory on disk. Of this amount of memory, 45 kilobytes can be attributed to classes relating to networking. As mentioned earlier, J2ME CLDC does not have these capabilities built in. However, compared to the Java J2ME CDC 1.0.1 platform, targeted at devices that are slightly less resource-constrained, in which the object serialization and reflection class files in respectively the `java.oi` and `java.lang.reflect` packages occupy approximately 200 kilobytes, we have a memory footprint reduction of approximately 40%. When the runtime is loaded into the virtual machine, but idle with no FROBs deployed on it, the total memory consumption is 139 kilobytes, of which 87 kilobytes are attributed to the Java virtual machine itself.

4.2 Indirection Overhead

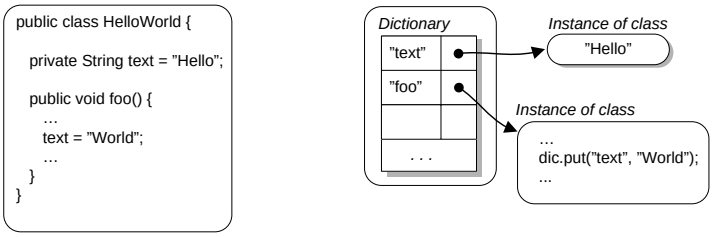
To measure the performance overhead due to the level of indirection we add within the FROBs, we conducted two experiments. First, we measured the raw performance overhead, i.e., the overhead of the indirection when invoking methods without functionality. Second, we measured the practical overhead, i.e., the overhead of the indirection when invoking methods with functionality from our scenario. In both cases, the results were compared to invoking the equivalent methods directly, i.e., without indirection.

Our experiments indicate that the level of indirection causes a raw overhead of a factor of 5-8 when compared to performing direct invocations. This is attributed to the actual lookup in the `java.util.HashMap`, which is used as a foundation for our dictionary.

However, our experiments also indicate that (when invoking methods containing functionality used) in the audio streaming scenario, this overhead becomes insignificant compared to the total time spent executing the functionality. For instance, when invoking methods with functionality from the scenario the overhead only accounts for approximately 0.5%, which, we believe is, insignificant at this level of prototyping.

4.3 Fine-Grained Representation Overhead

Representing the components using fine-grained behavioral objects stored in a dictionary induces an overhead compared to traditional object-oriented representations, where the functionality of the behavioral object would typically be represented as instance methods on some class as illustrated in Figure 14.



(a) Object-oriented representation

(b) Fine-Grained behavioral representation

Fig. 14. Component representations

To measure this overhead, we built different sets of components from our scenario using (a) our fine-grained behavioral object representation and then (b) a traditional object-oriented representation; we then compared both.

Whereas the overhead from our fine-grained behavioral object representation in terms of total size of class files on the disk is roughly a factor 2, the overhead on the heap size is roughly a factor of 8. For instance, compared to the 56 bytes of an object-oriented representation, the fine-grained behavioral representation of the audio provider takes 432 bytes. The reason for this overhead is mainly due to the use of a dictionary (which in itself accounts for 120 bytes when having no entries). In addition, each entry in the dictionary also requires a `java.lang.String` as key, which also accounts for heap space – an empty instance of `java.lang.String` requires 40 bytes with the J2SE virtual machine we employ. The overhead caused by our intensive use of the heap, instead of the call stack, is a well known issue that can be mitigated through various techniques such as escape analysis [9].

4.4 Resource Profile Generation Overhead

Conceptually, generating resource profiles for a behavioral object is done by the runtime. The FROB runtime does actually measure bandwidth by counting the number of bytes taken up by the events that the behavioral object publishes. However, because J2ME does not offer the possibility of controlling resources (CPU and memory) through the virtual machine, we had to augment the KVM (J2ME CLDC virtual machine) with functionalities to be able to count executed bytecodes and memory usage. The virtual machine has been extended in order to count the number of bytecodes executed and the number of bytes allocated in memory for a given behavioral object. Basically, two counters were added in the virtual machine: a *bytecodeCounter*, which is incremented everytime a bytecode is executed, and a *memoryCounter*, which is incremented by the size (in bytes)

of every memory allocation. These counters are reset and queried by two Java classes extending the CLDC API.

The size of the KVM shows a negligible increase of 0.3% (for a total of 102 kilobytes) and the growth of the API is 0.5% (for a total of 147 kilobytes). Note that our modified KVM, compiled on linux (gcc 3.4.5) and romized (CLDC API inlined in the executable) is only 203 kilobytes large after having stripped unnecessary sections out of the object file and becomes thuns less than 0.5% larger than the orginial one.

As counters are incremented at runtime on every bytecode execution and memory allocation, the overall execution speed of an application is slower. We have tested the modified KVM with two types of applications that simulate the worst cases for both bytecode and memory accounting.

The first test program executes a very large number of times bytecodes that are translated into the least hardware instructions possible (`NOP` bytecodes), in order to maximize the cost of the bytecode counting itself. This situation results in the worst possible case, with respect to overall execution speed and bytecode counting. The second program allocates memory as often as possible, maximizing the number of times the virtual machine must count memory allocations, thus again, resulting in the worst case possible, with respect to overall execution speed and memory allocations counting. Results show that the modified KVM is at most 7% slower than the original one, which is very acceptable at this level of prototyping.

5 Related Work

We position in this section our FROB model with respect to some representative distributed systems and distributed computing models that are (1) *resource-cautious*, meaning that they were designed to generate a small footprint, (2) *resource-aware*, meaning that they provide hints about resource consumption and/or availability to the applications they host, or (3) *adaptive*, meaning that they provide adaptation mechanisms, e.g., dynamic code replacement, migration etc., to the applications they host.

Most distributed systems that are marketed for mobile devices focus on one or the other of these dimensions, while leaving the others out of scope. The challenge addressed by FROBs is precisely that of providing a computing model for the development of adaptive and resource-aware programs running on resource-cautious systems.

5.1 Resource-Cautious Models

Computing models currently considered by the industry for building mobile applications on resource-constrained devices, such as Java CLDC [33] and .Net Compact Framework [26], are merely descendants of programming models used to build traditional applications for more static environments. Neither the models, nor their runtime support, provide the constructed applications with adequate ability of combined adaptivity and resource-awareness.

To save memory footprint, runtime platforms for resource-constrained devices, such as Java J2ME CLDC [33] and OSVM [14], typically sacrifice *reflection*, usually key to adaptivity and resource-awareness. In fact, another consequence of the lack of reflection is the lack of general-purpose object serialization mechanisms obstructing the ability to communicate easily.

For very resource-constrained devices, the most notable research project following direction is TinyOS [18], which is centered around sensor networks. Like TinyOS, the FROB model is based on an event-based interaction scheme. TinyOS is resource-cautious 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.

TinyOS does not support adaptivity because once the code is linked and deployed on a device, it cannot be changed. The Maté [22] project addresses the replacement issue of TinyOS 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. [19], on the other hand, enables a whole image to be redeployed remotely on a node, which helps the dissemination at deployment time but does not enable non- interrupted runtime code replacement. In fact, the node loses its state as the deployment acts as a stop/restart with a new image. The FROB model, on the contrary, inherently provides fine-grained means for non-interrupted runtime code replacement.

More generally, Instead of proposing a scaled down variant of a modern computing model, the FROB model goes back to the roots of the seminal work of Dijkstra on guarded commands [12] and its derivatives [7]. The underlying idea is to divide a program into a set of atomic behavioral objects protected by predicates. A predicate determines the exact conditions under which a certain behavioral object can be executed. In the FROB context, resource profiles, as amount of resources (CPU, Memory, Bandwidth) that a behavioral object will presumably require, act as a condition to fulfil before starting the execution of the behavioral object.

5.2 Resource-Aware Models

There is no broadly accepted programming model for resource management, and, a fortiori, resource awareness. Several prototypes have been proposed, using Java (Standard Edition) as execution platform, such as the Aroma VM [36], KaffeOS [5] and the Multi-tasking Virtual Machine (MVM) [11], which all keep or extend the standard Java computing model and are not targetted at mobile devices.

An important obstacle that researchers in resource management typically face 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. It is planned that standard Java will provide such a facility in a future release [10].

On the other hand, resource management in the FROB model does not suffer the same problems that standard Java is facing. This is because the behavioral objects of a FROB are executed isolated from each other. Therefore, the resource management scheme is not only capable of reliably measuring resource consumptions of current executions, but also of usefully exploiting this data to predict upcoming executions, in a simple, resource-efficient way.

SEDA [37] promotes 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 model for mobile devices.

5.3 Adaptive Models

Many of the early distributed computing models provided a fully reflective and hence adaptive execution scheme [24, 3, 25, 1, 8]. None of those however was designed with resource-constrained devices in mind.

Like Emerald [20], the FROB computing model supports migration of running processes. However, unlike Emerald, the FROB model does not support migration of the process' thread. The FROB model is threadless, and thus maintains a loose coupling between the behavioral objects and the thread executing them. The threads are assigned to the execution of behavioral objects by the runtime in a time-slicing scheme. Given this time-slicing scheme, and the fact that, within any FROB, only one behavioral object at a time is executed, the checkpointing of the runtime state of a running process between two behavioral object executions is straightforward.

Other projects, such as [13, 28], have also addressed adaptivity – though still somewhat inflexible – with predefined service levels and infrastructure responsibility to actually initiate the possible changes. In SOS [16], the ability to reconfigure a node can lead to corrupt the consistency of the application, caused by intermodule dependencies and should be seen as a way to update an application slightly and for long term, and not as a way to adapt the quality of service of a node. In addition, unlike Contiki [13] the FROB platform 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 better can exploit remote resources.

From the adaptive and control flow perspective, our FROB model is close to the actor model [17, 1] (and more precisely its ActorSpace [2] variant with anonymous event-based communication, itself inspired by [21]).

Unlike many concurrent computing models [20, 4, 38], but just like the actor model, only one behavioral object at a time is executed by a FROB, and behavioral objects do not contain synchronization statements.[6] In particular, *remote procedure calls*, be them completely synchronous, or semi-synchronous through the use of futures [31] or promises [23], are precluded within behavioral objects.

[6] Besides the underlying issues of thread and CPU management, such statements (e.g., *wait*, *fork/spawn*) significantly complicate code upgrading, concurrency control, and resource-based reasoning.

If needed, they are programmed through events across several behavioral object executions.

There are, however, three important differences between the FROB and the actor model. 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 run-time management of object identities and memory allocation. 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, prevents casting errors and enables cheap fine-grained serialization.

6 Concluding Remarks

This paper presents a candidate model for computing on mobile resource devices. 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.

On the basis of a prototype running on the Java platform for resource-constrained devices, we demonstrated the feasibility of our model through a demanding example of ad-hoc mobile computing, and we also gathered some encouraging deployment figures from our implementations.

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. Also, it is not yet clear to us how to deal, within behavioral objects, with *loop* constructs that cannot be statically analyzed. Specifically, such constructs complicate prediction of resource consumption of a behavioral object, and may to some extend compromise liveness. One proposal would be to require loops that cannot be statically analyzed to be explicitly *unfolded*, such that each iteration is expressed in terms of an event published to the behavioral object itself, which could possibly be done with some language construct. Another aspect that also requires further investigation is some form of *lightweight authentication* that could be realistic to integrate within FROBs, for some applications might preclude FROBs from accepting a new behavior (or even an event) without appropriate accreditations.

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, Ole Lehrman Madsen, Boris Magnusson, Martin Odersky, and Reiner Schmidt.

References

1. G. Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, 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 (PPOPP'93)*, pages 23–32, San Diego, May 1993.
3. M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In *Workshop on Object-Based Distributed Programming (ECOOP'93)*, pages 152–184, Kaiserslautern, June 1993.
4. J. Armstrong. The development of Erlang. In *Proceedings of the 2nd ACM SIGPLAN international conference on Functional programming (ICFP'97)*, pages 196–203, Amsterdam, June 1997.
5. G. Back, W. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, October 2000.
6. S. Baehni, C. S. Chabra, and R. Guerraoui. Frugal event dissemination in a mobile environment. In *Proceedings of the ACM/IFIP/USENIX 6th International Middleware Conference (Middleware 2005)*, Grenoble, November 2005.
7. H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.
8. J.-P. Briot, R. Guerraoui, and K.-P. Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, September 1998.
9. J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 1–19, Denver, November 1999.
10. G. Czajkowski and L. Daynès. Multitasking without compromise: A virtual machine evolution. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*, pages 125–138, Tampa Bay, October 2001.
11. 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, November 2004.
12. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.
13. A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 1st IEEE Workshop on Embedded Networked Sensors*, Tamba Bay, November 2004.
14. Esmertec. *OSVM.* http://www.esmertec.com.
15. P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.
16. C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd International Conference on Mobile systems, applications and services (Mobisys 2005)*, pages 163–176, Seattle, June 2005.
17. C. E. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3), June 1977.
18. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *SIGOPS Operating Systems Review*, 34(5):93–104, December 2000.

19. J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd International Conference on Embedded networked sensor systems (SenSys 2004)*, pages 81–94, Baltimore, November 2004.
20. 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, February 1988.
21. W. A. Kornfeld and C. E. Hewitt. The scientific community metaphor. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(1):24–33, January 1981.
22. 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 (ASPLOS-X)*, pages 85–95, San José, October 2002.
23. 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 (PLDI'88)*, pages 260–267, Atlanta, June 1988.
24. C. V. Lopes and G. Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010, P9710047, Palo Alto, February 1997.
25. H. Masuhara and A. Yonezawa. An object-oriented concurrent reflective language ABCL/R3: Its meta-level design and efficient implementation techniques. In J.-P. Bahsoun, T. Baba, J.-P. Briot, and A. Yonezawa, editors, *Object-Oriented Parallel and Distributed Programming*, pages 151–165. HERMES Science Publications, Paris, 2000.
26. Microsoft. Microsoft .NET framework. http://www.microsoft.com/net.
27. Microsoft. *DCOM Technical Overview (Microsoft White Paper)*, 1999.
28. A. Mukhija and M. Glinz. A framework for dynamically adaptive applications in a self-organized mobile network environment. In *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - W2: DARES (ICDCS 2004)*, pages 368–374, Tokyo, March 2004.
29. OMG. *The Common Object Request Broker: Architecture and Specification*, February 1998.
30. K. Raatikainen. A new look at mobile computing. In *Proceedings of Academic Network for Wireless Internet Research in Europe (ANWIRE) Workshop*, Athens, May 2004.
31. J. Robert H. Halstead. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.
32. R. B. Smith and D. Ungar. Programming as an experience: The inspiration for Self. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, pages 303–330, Aarhus, August 1995.
33. Sun Microsystems. *Java 2 Platform, Micro Edition, Connected Limited Device Configuration (CLDC)*. http://java.sun.com/products/cldc.
34. Sun Microsystems. *Java 2 Platform, Standard Edition*. http://java.sun.com/.
35. Sun Microsystems. *Java Remote Method Invocation – Distributed Computing for Java (Sun Microsystems White Paper)*, 1999.
36. 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 2000)*, pages 163–164, Barcelona, June 2000.
37. M. Welsh and D. Culler. Overload management as a fundamental service design primitive. In *Proceedings of the 10th ACM SIGOPS European Workshop*, Saint-Emilion, September 2002.
38. A. Yonezawa and M. Tokoro. *Object-oriented concurrent programming*. MIT Press, Cambridge, 1987.