Pervasive Computing with Frugal Objects

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

Rachid Guerraoui, Jarle Hulaas, Maxime Monod, Jesper H. Spring
Ecole Polytechnique Fédérale de Lausanne (EPFL),
School of Computer & Communication Sciences,
CH-1015 Lausanne
Email: {rachid.guerraoui, jarle.hulaas, maxime.monod, jesper.spring}@epfl.ch

***Abstract*— This paper presents a computing model for resource-limited mobile devices that might be ubiquitously deployed in private and business environments. The model integrates 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 Frugal Objects (FROBs), that communicate through typed events and dynamically adapt their behavior according to notifications about changes in 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, battery and bandwidth. The behavior of a FROB is represented by a set of stateless first-class objects. Both state and behavioral objects are referenced through a level of indirection within 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 increase of the underlying FROB runtime.**

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

I. 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 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 used by the software, including services provided in the surrounding environment, 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.

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 calls for a computing model where several flows of control 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 internal 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. The RPC abstraction aims at promoting the porting of centralized programs into 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* (also called *promises*) only address the synchronization part of the problem. Mobile environments rather call for anonymous and one-way communication schemes.

Devising a computing model that, while obeying the above principles, remains simple to comprehend yet implementable on resource-constrained devices, is rather challenging.

Overview

This paper presents a computing model based on frugal objects, called *FROBs*, which are supposed to be deployed ubiquitously 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 Fig. **??**). A FROB can specify the type of events it can process, and how, through *behavioral objects* ((3) in Fig. **??**). 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 Fig. **??**). Upon receiving an event, the runtime matches it against the interface to determine whether to accept the event for further processing or reject it.

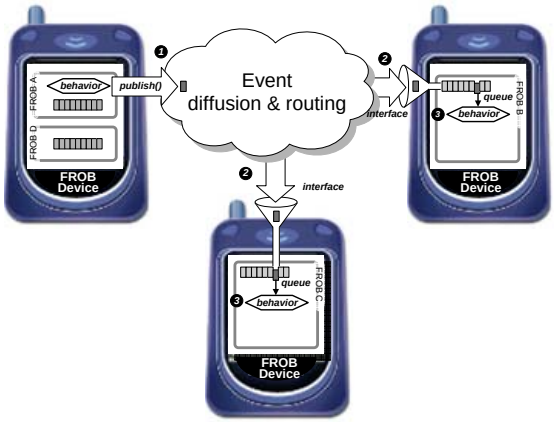

Fig. 1. Event-based interacting FROBs

- Besides preventing casting errors and acting as a filtering mechanism, our typed event model promotes a fine-grained serialization scheme that exploits the decentralized representation of a behavior, and its binding to event types.
- Key to supporting adaptivity with minimal underlying footprint is the stateless representation of a FROB behavior as a set of first-class objects 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.
- 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 and can be approximated a priori.
- 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, 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.

We report on a prototype implementation of our computing model on top of the Java J2ME CLDC platform [1] targeted at resource-constrained devices, including modifications made to the KVM virtual machine to experiment with the generation of resource profiles. The implementation adds negligible extensions to the memory footprint of the virtual machine and the API, but introduces a slowdown of 6-10% because of runtime resource profiling.

The structure of the rest of this paper is as follows. Sec. 2 overviews how to program with frugal objects. Sec. 3 details the FROB computing model, and Sec. 4 gives some further insights into the resource profiling. Sec. 5 discusses aspects of the prototype implementation, and Sec. **??** positions the FROB computing model with respect to existing work. Finally, Sec. 6 makes some concluding remarks.

II. Programming with Frugal Objects

A FROB conceptually consists of (Fig. 2): (1) an external interface made of event types and 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 of the FROB.

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

The event queue of the FROB (see (2) in Fig. 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 decentralized model of programming with multiple flows of control.

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 (and invoking the `getEventType()` method, which is mapped into subscriptions for event types that the behavioral objects are capable of handling.

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

FROBs are encapsulated entities that do not share state (i.e., entries in the dictionaries) – the behavioral objects always

Programming a Behavioral Object:

```
class DecodeAudio extends BehavioralObject {
    public Class getEventType(){
       return AudioEvent.class;
    }
    public void handleEvent(byte[] ba) {
       AudioEvent evt = deserialize(ba);
       byte[] raw = decodeAudioEvent(evt);
       byte[] playEvt =
          serialize(new playAudioEvent(raw));
       publish(playEvt);
       ...
    }
    private byte[] decodeAudio(AudioEvent evt) {...}
    private AudioEvent deserialize(byte[] ba) {...}
    private byte[] serialize(Event evt) {...}
}
```

Programming a FROB:

```
class AudioPlayer extends FROB {
    public void initialize() {
       setQueueSize(100);
       dictionary.put("init", new InitAudioPlayer());
       dictionary.put("decode", new DecodeAudio());
       dictionary.put("play", new PlayAudio());
       dictionary.put("counter", new Integer(0));
       ...
    }
}
```

A Conceptual View of a FROB:

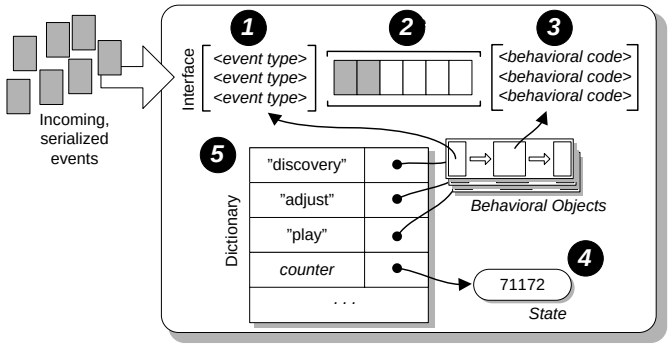

Fig. 2. **Example.** A FROB, `AudioPlayer`, processes audio samples when received through typed events, this processing includes decoding the audio sample into a raw byte stream, `DecodeAudio`, which is then sent to the behavioral object responsible for the actual audio playing.

run isolated from each other. This combination eliminates the need for synchronization on entries in the dictionary.

III. Frugal Object Properties

A. Typed Events

Events are the basic entity to which FROBs react: the reception of an event is the only means by which a behavioral object in a FROB is executed. Events 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 expensive in terms of stack management and over-provisioning of stacks, as well as locking mechanisms.

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.

B. 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, 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 (Fig. 3).

We exploit the very fact that communication between

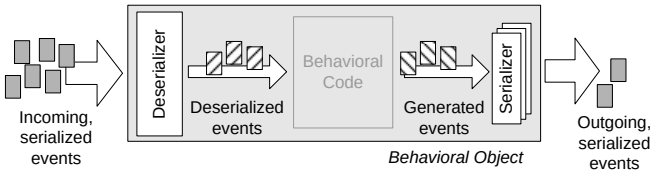

Fig. 3. Behavioral object with deserializer and serializers

FROBs occurs through typed events. As mentioned earlier, each behavioral object is bound to the typed event it can handle, and its execution is solely triggered by a particular typed event. In other words, the (de-)serialization capabilities required for each behavioral object are thus 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. We circumvent this potential overhead by simply transparently sharing these capabilities between behavioral objects based on the same event type, and thereby only loading a single instance of the functionality.

C. 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 objects: behavioral objects. This fine-grained granularity allows for flexible modifications 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 (Fig. **??**). In fact, this also enables behavioral objects to cause their own replacement.

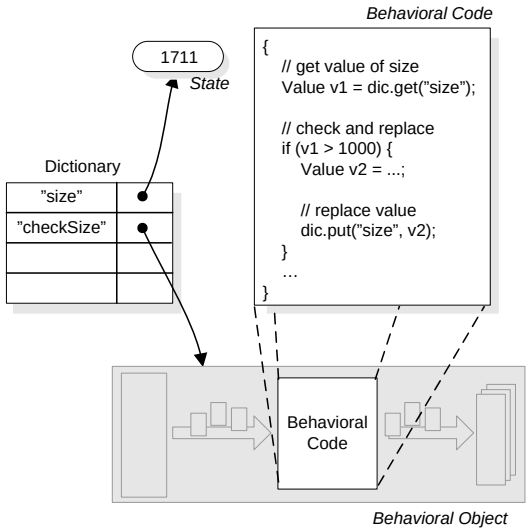

Fig. 4. Dictionary with state and behavioral objects

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

D. 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 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. These two mechanisms combined with the time-slicing scheme gives the FROB runtime explicit *control points* between executions, i.e., the FROB runtime has total control over the FROBs between each granted time-slice. 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, if possible with respect to available resources. The resource requirements of these individual behavioral objects are thus limited in terms of actual resource

amount needed and their usage duration. These requirements are associated to each behavioral object expressed in a resource profile used by the runtime. 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 within 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.

IV. Resource-Profiling

The FROB runtime constantly monitors the availability of internal resources such as CPU, memory, bandwidth etc (Fig. 4). 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.

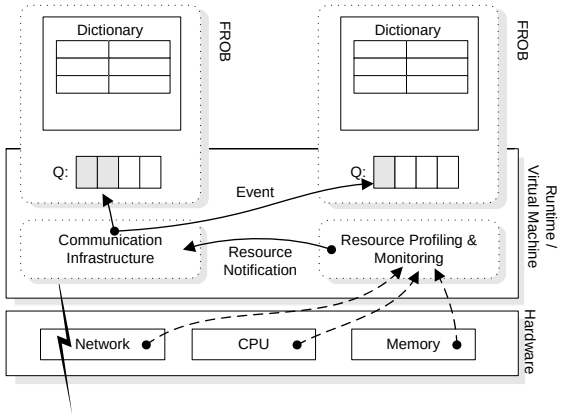

Fig. 5. Resource profiling and monitoring in the FROB runtime

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 of the behavioral objects, and are attached to them 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. The FROB runtime uses a best-effort strategy to determine if enough resources are available to execute a behavioral object. In fact, there is no guarantee that the behavioral object can run to completion without experiencing resource-related errors. If not enough resources are available, 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. We considered the following strategies

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. In particular, this can be cause by 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.

V. Implementation

A prototype of FROBs have been implemented on top of the Java J2ME virtual machine, which was designed for resource-constrained uniprocessor embedded devices. Our implementation consists of modifications made to the KVM virtual machine as well as a small API exploiting these modifications. The size of the compiled KVM virtual machine shows a negligible increase of 0.3%, and the growth of the API extensions account for 0.5% compared to the default J2ME CLDC API.

A. Class-Unloading

The mechanism of loading classes is per default implemented in the Java virtual machine. Unloading of classes is, however, not possible[1]; only instances of classes can be unloaded. While this might be sufficient in a resource rich environment, such class definitions might quickly add up in an environment characterized by few available resources and frequent dynamic changes involving loading of new classes.

In our implementation, the FROB API contains a method to manually have a class definition removed from the class table at the next garbage collection. For this purpose, KVM was configured with a non-compaction garbage collector to prevent the class elements from being allocated in immortal memory and instead be represented as normal objects subject to garbage collection. Currently, the process of unloading a class is not safe, i.e., there are no security checks that the class being removed is no longer in use.[2]

B. Resource Profiling

Our resource profiler has been implemented as native extensions to the virtual machine. In the current implementation, the resource profiler accounts for memory and CPU-cycles, the latter represented in terms of number of bytecodes, in addition to deducing an approximate energy consumption following from these.

The resource profiles are generated at runtime when each behavioral object is executed. Counters have been added to the native thread to (1) count the number of bytecodes being executed in a behavioral object, and (2) sum up the total number of bytes being allocated on the heap. These counters are reset every time the thread starts to execute a behavioral object, and are read when the execution is done, except if execution is aborted following an exception.

Energy (in μJ) consumption is deduced from a categorization of bytecodes and a per-bytecode energy consumption model [2], i.e., a table containing for each bytecode a corresponding energy cost (aggregating CPU instructions and memory energy costs).

Having generated a resource profile containing memory, CPU and energy for a given behavioral object, the profile object is attached to the behavioral object, and is available through the `getResourceProfile()` method on the resource profile object.

As counters are incremented at runtime on every bytecode execution and memory allocation, the overall execution speed of an application is reduced. Using CaffeineMark 3.0 benchmark [3] for embedded devices, we have measured a speed reduction caused by the dynamic profiling of 9.23% as depicted in Fig. 5. Using similar benchmarks such as JGrande 2.0 [4] and JOlden [5], we experienced speed reductions of respectively 6.46% and 9.86% following the dynamic profiling, which we believe is very acceptable at our prototyping level.

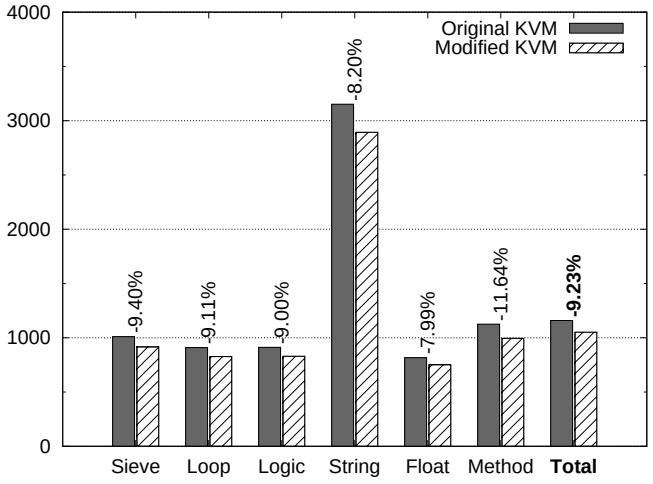

Fig. 6. CaffeineMark 3.0 benchmarks showing relative performance decrease (in points) imposed by adding dynamic resource profiling of behavioral objects to the KVM.

C. Resource Monitoring

For resource monitoring and notification, our current implementation is limited to only monitor changes in the level of available memory. Resource levels are monitored following execution of behavioral objects. Using a policy-based approach, the resource monitoring mechanism publishes events following the detection of significant changes in resource availability according to threshold values (at each extreme of the spectrum) defined in the policy.

VI. Related Work

We position here our model with respect to some alternative models designed to be to be (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.

In short, 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.

A. Resource-Cautious Models

Computing models currently considered by the industry for building mobile applications on resource-constrained devices, such as Java J2ME CLDC [1] and .Net Compact Framework [6], are merely descendants of programming models used to build traditional applications for more static environments.

[1]Except when using the rather heavyweight custom classloader framework present only in J2SE.

[2]However, for the purpose of illustrating the usefulness of class-unloading, this limitation implies that the programmer must ensure that the classes contained in and used exclusively by the behavioral object can only be removed once the behavioral object has been removed from the dictionary of the FROB.

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 [1] and OSVM [7], typically sacrifice *reflection*, usually key to adaptivity and resource-awareness. [3]

For very resource-constrained devices, the most notable research project is TinyOS [8], which is centered around sensor networks. TinyOS supports limited adaptivity because once the code is linked and deployed on a device, it cannot be changed. The Maté [9] project addresses the replacement issue of TinyOS by providing a virtual machine 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. Deluge [10], 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.

B. 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 [11], KaffeOS [12] and the Multi-tasking Virtual Machine (MVM) [13], which all keep or extend the standard Java computing model and are not targetted at mobile devices.

The behavioral objects of a FROB are typically small units of executions without synchronization primitives, and 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 [14] 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.

C. Adaptive Models

Many of the early distributed computing models provided a fully reflective and hence adaptive execution scheme [15], [16], [17], [18], [19]. 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 [21], [22], [23], have also addressed adaptivity with predefined service levels or infrastructure responsibility to actually initiate the possible changes. In SOS [24], 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 [21] 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 [25], [15] (and more precisely its ActorSpace [26] variant with anonymous event-based communication, itself inspired by [27]).

Unlike many concurrent computing models [28], [20], [29], 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. In particular, *remote procedure calls*, be them completely synchronous, or semi-synchronous through the use of futures [30] or promises [31], are precluded within behavioral objects. If needed, they are programmed through events across several behavioral object executions.

There are, however, important differences between the FROB and the actor model. 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. Also, 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.

VII. Concluding Remarks

We presented a computing model, *FROBs*, for programming adaptive, resource-aware applications on resource-constrained mobile devices. Frugal Objects use a set of loosely coupled pieces of logic, represented as behavioral objects, and a level of indirection to enable reflective adjustment of functionality over the lifetime.

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 [32] and its derivatives [33]. The underlying idea is to divide a program into a set of behavioral objects protected by predicates. A predicate determines the exact conditions under which a certain behavioral object can be executed. In the

[3] In fact, another consequence of the lack of reflection is the lack of general-purpose object serialization mechanisms obstructing the ability to communicate easily.

FROB context, resource profiles, as amount of resources (CPU, memory, bandwidth) that a behavioral object will presumably require, acan typically act as a condition to fulfill before starting the execution of the behavioral object.

By requiring complete encapsulation of the behavioral objects, including their required deserializer and serializer, adaptation to environment changes becomes a question of changing the set of loaded behavioral objects, and since these can be completely unloaded, and thus completely release any bound resources.

Through the introduction of profiling of behavioral objects, the resource consumption of execution can be determined, and can lay the foundation for adapting the set of behavioral objects in order to encompass current resource availabilities.

An initial prototype implementation of the base functionality of FROBs shows that the extensions to the virtual machine and API only increase the footprint slightly, and that the cost of doing runtime resource profiling introduces a slowdown of 6-10%.

REFERENCES

[1] *Java 2 Platform, Micro Edition, Connected Limited Device Configuration (CLDC)*, Sun Microsystems, http://java.sun.com/products/cldc.

[2] S. Lafond and J. Lilius, "An energy consumption model for an embedded java virtual machine," in *ARCS '06: Proceedings of the 19th International Conference on Architecture of Computing Systems*, Frankfurt, March 2006, pp. 311–325.

[3] *CaffeineMark 3.0 for Embedded Devices*, Pendragon Software Corporation, http://www.benchmarkhq.ru/cm30/info.html.

[4] C. Daly, J. Horgan, J. Power, and J. Waldron, "Platform independent dynamic java virtual machine analysis: the java grande forum benchmarking suite," in *Proceedings of the joint ACM-ISCOPE conference on Java Grande*, Palo Alto, 2001, pp. 106–115.

[5] B. Cahoon and K. S. McKinley, "Data flow analysis for software prefetching linked data structures in java," in *PACT '01: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Barcelona, September 2001, pp. 280–291.

[6] Microsoft, "Microsoft .NET framework," http://www.microsoft.com/net.

[7] *OSVM*, Esmertec, http://www.esmertec.com.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," *SIGOPS Operating Systems Review*, vol. 34, no. 5, pp. 93–104, December 2000.

[9] 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)*, San José, October 2002, pp. 85–95.

[10] 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)*, Baltimore, November 2004, pp. 81–94.

[11] 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)*, Barcelona, June 2000, pp. 163–164.

[12] 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.

[13] G. Czajkowski, S. Hahn, G. Skinner, P. Soper, and C. Bryce, "A resource management interface for the Java platform," *Software Practice and Experience*, vol. 35, no. 2, pp. 123–157, November 2004.

[14] 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.

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

[16] 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)*, Kaiserslautern, June 1993, pp. 152–184.

[17] J.-P. Briot, R. Guerraoui, and K.-P. Lohr, "Concurrency and distribution in object-oriented programming," *ACM Computing Surveys*, vol. 30, no. 3, pp. 291–329, September 1998.

[18] C. V. Lopes and G. Kiczales, "D: A language framework for distributed programming," Palo Alto, Tech. Rep. SPL97-010, P9710047, February 1997.

[19] H. Masuhara and A. Yonezawa, "An object-oriented concurrent reflective language ABCL/R3: Its meta-level design and efficient implementation techniques," in *Object-Oriented Parallel and Distributed Programming*, J.-P. Bahsoun, T. Baba, J.-P. Briot, and A. Yonezawa, Eds. Paris: HERMES Science Publications, 2000, pp. 151–165.

[20] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-grained mobility in the Emerald system," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 109–133, February 1988.

[21] 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.

[22] 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)*, Tokyo, March 2004, pp. 368–374.

[23] P. Costa, G. Coulson, C. Mascolo, L. Mottola, G. P. Picco, and S. Zachariadis, "A Reconfigurable Component-based Middleware for networked Embedded Systems," *Journal of Wireless Information Networks*, 2006, to appear.

[24] 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)*, Seattle, June 2005, pp. 163–176.

[25] C. E. Hewitt, "Viewing control structures as patterns of passing messages," *Journal of Artificial Intelligence*, vol. 8, no. 3, June 1977.

[26] 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)*, San Diego, May 1993, pp. 23–32.

[27] W. A. Kornfeld and C. E. Hewitt, "The scientific community metaphor," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 11, no. 1, pp. 24–33, January 1981.

[28] J. Armstrong, "The development of Erlang," in *Proceedings of the 2nd ACM SIGPLAN international conference on Functional programming (ICFP'97)*, Amsterdam, June 1997, pp. 196–203.

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

[30] J. Robert H. Halstead, "MULTILISP: a language for concurrent symbolic computation," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 4, pp. 501–538, October 1985.

[31] 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)*, Atlanta, June 1988, pp. 260–267.

[32] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, August 1975.

[33] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, "Programming languages for distributed computing systems," *ACM Computing Surveys*, vol. 21, no. 3, pp. 261–322, September 1989.