FRUGAL MOBILE OBJECTS

Benoît Garbinato
Université de Lausanne,
Ecole des HEC,
CH-1015 Lausanne
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
{rachid.guerraoui, jarle.hulaas, maxime.monod, jesper.spring}@epfl.ch

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, 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, 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

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 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.

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 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. 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* (also called *promises*) only address the synchronization part of the problem. Mobile environments rather call for anonymous and one-way communication schemes.

Devising a robust 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 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. 1). A FROB can specify the type of events it can process, and how, through *behavioral objects* ((3) in Fig. 1) – also called *actions*. Each such object is 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 Fig. 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, our typed event model makes it possible to adopt a fine-

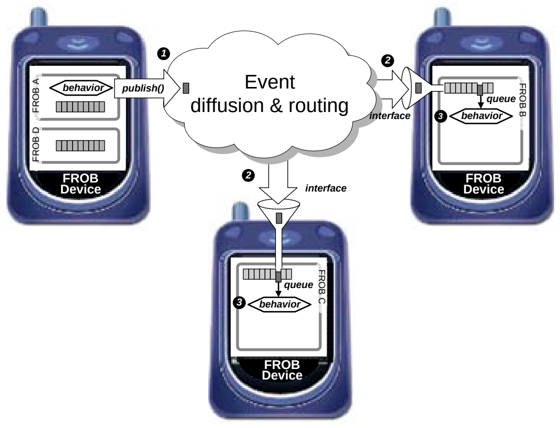


Figure 1. Event-based interacting FROBs

grained serialization scheme that exploits the decentralized representation of a behavior, and its binding to event types. Our serialization mechanism does thus not rely on a general (footprint greedy) 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 also 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 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.

2. The FROB Computing Model

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.

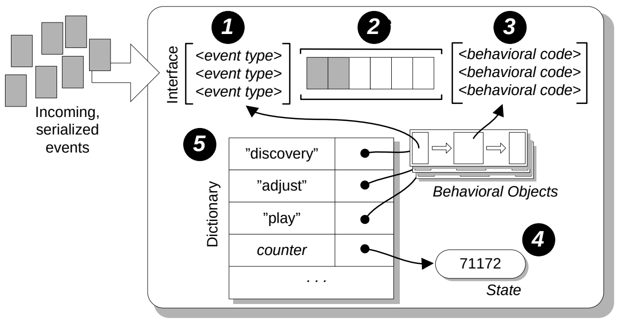


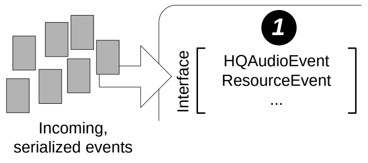
Figure 2. Conceptual view of a FROB

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 Fig. 2). The notion of dictionary is similar to that of *slots* in the Self language; a slot can contain either state or code.

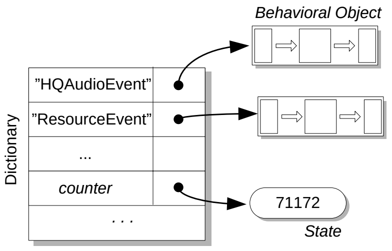
The event queue of the FROB (see (2) in Fig. 2) is not contained in the dictionary and is under the sole control of the FROB runtime, i.e., the FROB has no direct access to it and its only way to consume events is by having adequate behavior capable of handling the events. This enforces a decentralized 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, which is mapped into subscriptions for event types that the behavioral objects are capable of handling. This is illustrated in Fig. 3, which shows how the event types defined in the external interface (Fig. 3(a)) corresponds to actual behavioral objects present in the dictionary (Fig. 3(b)).

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



(a) Event types in external interface



(b) Corresponding behavioral objects in dictionary

Figure 3. Representing the external interface using behavioral objects from the dictionary (excerpt of Fig. 2)

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.

2.1 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.

2.2 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. 4).

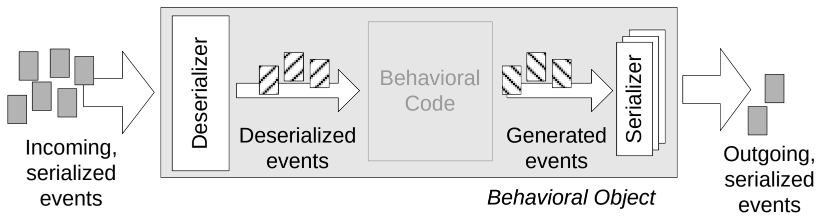


Figure 4. Behavioral object with deserializer and serializers

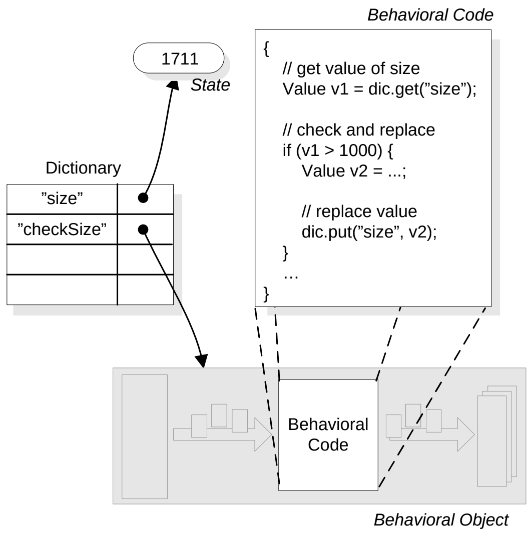
We exploit the very fact that communication between FROBs occurs through typed events. As mentioned earlier, each behavioral object is bound to the typed event it can handle, and its execution is solely triggered by a particular typed event. In other words, the (de-)serialization capabilities required 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.

2.3 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.



```
{
   // get value of size
   Value v1 = dic.get("size");

   // check and replace
   if (v1 > 1000) {
      Value v2 = ...;

      // replace value
      dic.put("size", v2);
   }
   ...
}
```

Figure 5. Dictionary with state and behavioral objects

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. 5). 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 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.

2.4 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.

2.5 Resource-Profiling

The FROB runtime constantly monitors the availability of internal resources such as CPU, memory, bandwidth etc (Fig. 6). 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.

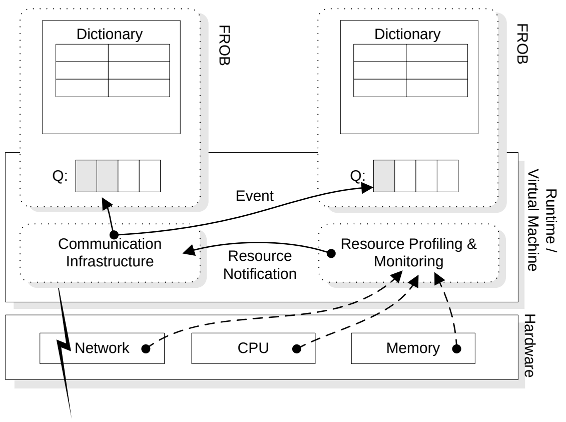

Dictionary
FROB
Dictionary
FROB
Q:
Q:
Event
Runtime / Virtual Machine
Communication Infrastructure
Resource Notification
Resource Profiling & Monitoring
Hardware
Network
CPU
Memory

Figure 6. 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.

3. 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.

Further experiments are needed and many FROB aspects need to be refined. Among these aspects, our prototyping revealed that, not surprisingly, the event-based style of programming combined with the manual managing of named entries in the dictionary put some complexity on the programmer. There is a clear need for high-level abstractions for code reuse and dependency management. Further research is for instance 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.

The *atomicity* of the behavioral objects is also not entirely clear to manage. Even if our behavioral objects cannot currently be interrupted by the runtime, we need to envisage interruptions schemes, in particular those due to resource shortages. Implementing atomicity with small footprint is however not trivial. 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.