Distributed Reliable Object-Oriented Programming (DROOP)

P. Eugster

November 3, 2003

Abstract

The *Simple Object-Oriented Concurrent Programming (SCOOP)* model proposed by Bertrand Meyer and illustrated through the Eiffel programming language is a simple yet powerful model for concurrent programming. In this paper, we analyze the applicability of the SCOOP model to physically distributed systems manifesting transient and permanent failures. We suggest additions to the basic SCOOP model in order to cope with such failures, coining the term *Distributed Reliable Object-Oriented Programming (DROOP).*

1 SCOOP and Distribution

This section first summarizes the SCOOP model, and then evaluates it roughly in the context of distributed programming.

1.1 Summary of SCOOP

With the SCOOP concurrency control model [9], *objects* are assigned (implicitly or explicitly) to *processors*. Latter notion captures well active entities executing the code of objects, and hence the application itself. These entitites can range from lightweight OS threads, over OS processes, to CPUs themselves. Operations called on a target object are executed by the processor associated with that object. The distinction between a call to a "local" object, i.e., an object associated with the same processor as the caller, and a call to an object governed by a distinct processor, becomes then necessary. The distinction is made by tagging corresponding definitions with the keyword `separate`, the only syntactic addition to the language required by the scheme.

1.2 Evaluation in the Face of Distribution

The clear separation of execution flow management and executed software text, just like the distinction between the judicial and the legislative (with the effective CPU's taking the role of the executive), seems very plausible and appealing at the same time. The use of a single primitive speaks for the simplicity of the solution.

Coming from a distributed systems context, the main reservation one might have regarding the core mechanism is the seeming absence of distinction between concurrent and distributed execution. In fact, concurrency at a thread level (intra-process), a process level (inter-process), and even host level (network) do indeed manifest similarities. They can even be considered as responding the same rules, but only as long as *failures* are not considered. The likelihood that a pipe between two OS processes fails might be negligible, but is it safe to say as much about the probability that a process fails? The eventuality of the failure of a host, or of the communication with it, especially when targeting realistic large scale and distributed settings (i.e., Internet-scale) is not to be neglected. Transient network failures, e.g., due to congestion, can be counteracted by repeatedly sending the same information, but can lead to unbounded delays, hampering the overall performance of a distributed application, and ultimately, its correctness. Permanent failures on the other hand, are yet more wicked, as they can not be distinguished from transient ones in a practical distributed environment such as the Internet [5].

Even by granting only a weak probability to process or host failures, the shier size (in terms of involved hosts) and load of modern distributed applications makes these probabilities accumulate to a considerable threat. It is this size also, that introduces a strong *dynamism* into such settings, as one can expect participants to join and leave at runtime, sensitive events which themselves are associated with failures.

Different ways exist to obtain reliable distributed applications in the face of failures and dynamism. Two of the best known paradigms are (1) transactions, as a mechanism enforcing *concurrency*, and (2) (software) replication, as a solution for ensuring *availability* of components.[1] How the concurrency model of SCOOP relates to those paradigms, or how the SCOOP model could be enhanced with those paradigms, will be discussed in the next section. Last but not least, *asynchrony*, is an essential ingredient to a distributed programming model as it permits to reduce dependencies between components and hence potentially sustain higher failure rates. This issue will also be discussed in detail in the next section.

2 Towards Reliable Distributed Programming

In the following, several concerns about the applicability of the SCOOP model in the face of physical distribution are detailed. These concerns are less about the core model; the presented thoughts are more centered around the implications of the model. Proposals for improvement are provided where possible, and are all mostly orthogonal.

[1]This is illustrated by the wide application of so-called "application servers" in industrial software development, which precisely provide such features and greatly hide cumbersome details of their implementation.

2.1 Failures

As outlined in the first section, distribution entails specific failure patterns. These should be reflected in an application. In a setting in which all remote communication takes place through remote operation call, these remote calls should reflect the possibility of failures, for instance network failures resulting from temporary network congestion, through specific exceptions. This is independent from the semantics (and the implementation) of the remote communication, i.e., best-effort, at-most-once, as ultimately, target processes can fail. A poor (but at least some) way of expressing this possibility [3] is given by the Java binding of CORBA, where remote calls can raise exceptions, however of a specific type which does not imperatively have to be handled by the caller. This gives the impression that remote calls are identical to local ones. Better practice can be observed in Java RMI, where features of remotely callable object types must explicitly indicate the possibility of returning exceptions related to remote communication in corresponding operation signatures, and callers are obliged to handle these exceptions (checked at compilation).

In the context of the SCOOP model, one could easily add a rule that, whenever a `separate` object is called, the caller has to deal with a specific type of exception reflecting distribution-related problems. The necessity for dealing with failures upon invocation of separate objects seems to be somewhat already given by the possibility of aborting remote calls because of express messages. Special care would be required in the context of wait-by-necessity; not waiting for a remote command call to terminate means losing the ability of knowing whether the call has completed, or whether a failure has occurred (see Section 2.3).

The possibility of failures should also be considered when reserving objects. As a feature is only executed once all its arguments declared formally as `separate` are reserved, one can enter a blocking situation if any of these objects has failed, or communication with such an object exhibits failures.

2.2 Dynamism

The use of a configuration file seems interesting for bringing together the judiciary and the executive (see above). In any distributed scenario, a form of bootstrapping is required for remote entities to meet in the first place, and a configuration file can be used for that. However, the dynamism of modern distributed systems must be allowed more room.

More precisely, configuration files can be useful for defining at a physically local scale what processors should be associated with, and can help physically remote parties make first contact. A solution where remote hosts and remote interaction are hardwired does in general however not embrace the dynamic nature of modern distributed applications. A server may at some point run on a particular machine, and later on another. Changing a configuration file, and possibly even several ones (as distributed components might not have access to the same file system), every time the setting changes seems overly cumbersome. In particular, at the beginning of the execution of a distributed application, the host of a crucial server might not even be known at all.

The configuration file should not be the first choice for finding remote parties. A well-known, and widely used, concept for building reliable distributed object systems is that of a lookup service, such as a name service (comparable to white pages) or a trade service (comparable to yellow pages). Such a service is easily built, and provided to the application as a set of classes. It is typically the address of such a service which can be put in a configuration file.

The suggestion for *separate creation* in the software text, as introduced on page 970 in [9] and discussed on page 974 goes in that direction. The entries in the software text however seem a bit redundant to those of the configuration file. Surely reasonable effort would suffice to provide a solution which has at least the practical benefits of current practical solutions, and by being tailored to SCOOP, can even achieve a level of language-support not reached previously.

2.3 Concurrency

The concurrency scheme offered by SCOOP is interesting, and intriguing at the same time for distributed systems practitioners.

The consistency model of SCOOP is strongly based on wait-by-necessity and its definition on pages 989-991 in [9]. It requires that in a series of invocations on the same separate object, a query can only proceed if all preceding calls have completed. In distributed systems, wait-by-necessity is usually further stretched, leading to waiting for the return value of a remote call only if one needs that value effectively and immediately in the calling code. The goal is to return, upon a remote query call, an object representing the return value (e.g., some form of proxy), but only "populating" that object once the value has been effectively received from the callee site. As long as the object is not used, e.g., queried itself in the context of further processing, one has no reason to block if the object is not ready. As such, this mechanism hence surely makes sense mostly with queries, but can also be used with commands which are sometimes pictured as having implicit return values, representing the success of their remote execution.

With such a scheme, one could hence very well proceed even after a query, on the client side. In [9] this possibility is pointed out, but said to require a boolean attribute for proxies to declare when the value is ready or not. This is useful in practice, but not a necessity. The goal can be simply to delay the use of the return value of a query as much as possible in the software text, in the hope that at execution the value will be available until then. If not, the client has to nevertheless start waiting, as the value is needed (which accounts for the name of this paradigm). Any invocation of the non-ready value hence implicitly leads to blocking. The main argument against wait-by-necessity is, as hinted in Section 2.1, the reaction to failures. A remote call whose "result" (simply a return signal, or a return value) is not used (immediately), can not indicate through an exception any failure in completing (whether due to a distribution-related failure or "logical" failure).

It appears that there is further room for increasing concurrency by focusing on effective (and possibly different flavors of) consistency requirements. The model suggested currently, i.e., a target object executes only operations issued by a single client object at a time, and that client object can issue several consecutive calls by "reserving" the target object, boils down to a form of *strict consistency* [11]. This model, though surely useful, allows only little concurrency, easily leads to blocking, and imposes constraints stricter than what many shared objects/their applications in fact require. Examples of weaker semantics are given by sequential consistency [8], causal consistency [1], or linearizability [7]. These can be implemented with a higher degree of concurrency, and hence better performance, but giving individual clients the same view of the execution as if they were the only ones. The fact that the distinction between query and command operations in such consistency models is already promoted by Eiffel, cries for a closer investigation of the wide body of research behind such consistency models in the face of the SCOOP model.

Investigating consistency models mainly affects the interplay of concurrent clients acting on the same object. Investigating the client and server side, also possible in isolation, makes even more sense when having a second look at consistency models. While on the callee site, intuition would suggest that two operations of a same object, whose bodies and preconditions do not involve the same attributes could be executed concurrently, the reservation semantics on the callee site more importantly could be replaced or supplemented by transactional, "all-or-none", semantics for the set of calls (possibly a choice made by library calls) made in the context of an operation.

Such a variant would also embrace fault tolerance. Since a client can fail, and such a failure can block a reserved callee object forever, a callee object has to observe caller objects with timeout mechanisms. Due to the unpredictable nature of asynchronous systems such as the Internet [5], such a timeout can however also expire without the client having effectively failed, leading to the sudden abortion of an operation, and leaving the computation of the surrounding routine in an intermediate state. In particular, by making use of wait-by-necessity, the state (i.e, which previous operations have completed, and which have not) can not be determined. With transactional semantics, the (partial) effects of such a routine could be made undone. Such a scheme could be implemented already with the current strict consistency model.

2.4 Availability

The paradigm of choice to deal with failures of critical objects is replication [2], which builds on so-called group communication [12]. Any sensitive environment for reliable distributed programming should provide access to at least a limited form of group communication.

Many group communication systems have been built throughout the past, say, 20 years; more recently, also object-oriented ones have appeared (e.g., [6]). Such services are usually available through libraries. A common technique in object systems exists in using so-called "group proxies", which represent a group of replicas of an object rather than a singleton object, and encapsulate the more complex communications protocols used for group communication. Such group proxies could also be added to offer, optionally, increased fault tolerance for critical components of distributed applications.

2.5 Asynchrony

Though an asynchronous mode for computing remote operations based on wait-by-necessity is presented in [9], the only means of communicating in SCOOP seems to be nonetheless bidirectional, one-to-one client/server communication, as suggested also by the title.

The need for more inherently asynchronous communication, and in particular, one-to-many interaction ("multicast") has been recognized in the context of Eiffel, as illustrated by [10]. That proposal suggests a form of type-based publish/subscribe interaction, where interests are expressed by subscribers on types of events. Those events, like in most schemes, are viewed as tuples, without operations. A subscription is issued by registering a callback procedure, which takes as argument an instance of the tuple type.

The proposed scheme makes no mention to content-based filtering, based on "properties" of event objects, while virtually all industrial engines, in order to faithfully implement standardized API's such as the Java Message Service or the CORBA Notification Service, include such support. The scheme obtained without content-based filtering could be qualify as "(type) broadcast", as opposed to a " (type) multicast". Type multicast is admittedly in general difficult to achieve in combination with type-safety and encapsulation, by supporting the distributed application of filters expressed on properties of event objects [4].

The question that comes to mind, is whether one could not use the concept of precondition, inherent to Eiffel, precisely to express content-based interests. An alternative to viewing events as instances of specific types could be to picture them as specific calls, i.e., calls of command features on proxies created explicitly for given types, representing all "subscribers" of that type. Such calls would then namely be performed on all objects of that type, which previously subscribed. An event, making the correspondence between publishers and subscibers, would then be defined by the called type and feature, and the values of the arguments of that call. The set of target subscribers which subscribed to that type by registering a callback object of that very type, and whose respective preconditions are fulfilled by the arguments, whould receive the event by seeing the corresponding feature called on their callback objects.

Such specific proxies would hence be a particular form of the group proxies presented above, or could possibly appear in a form unified with those.[2]

[2]Intuition suggests that such a marriage should be possible, as the concept of publish/subscribe is in fact an offspring of group communication, with weaker semantics and targeting at a larger scale.

3 Summary and Conclusions

SCOOP is indeed a very interesting model for concurrent programming. What distinguishes distributed programming from classic concurrent programming is mainly that of physical distribution, and the imposed consideration of failures. This report has presented various improvements to reflect such failures in Eiffel/SCOOP — further ones could be thought of. The remaining guestion is probably that of the gained benefits of these individual benefits in comparison with their respective intrusiveness. Answering this question is strongly related to answering the question of the level of failure-awareness and -tolerance that is most appropriate for SCOOP. To not break with the goal of keeping SCOOP simple yet general, one might retain only proposals (or parts of) in mind which are implementable as libraries, and view them as accessible in the context of a particular instantiation of SCOOP, for distributed reliable object-oriented programming ("DROOP").

References

[1] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. Causal Memory: Definitions, Implementations, and Programming. *Distributed Computing*, September 1995.

[2] K.P. Birman, T. Joseph, T. Raeuchle, and A. El Abbadi. Implementing fault tolerant distributed objects. *IEEE Transactions on Software Engineering*, 11(6):39–59, February 1985.

[3] P.Th. Eugster and S. Baehni. Abstracting Remote Object Interaction in a Peer-to-Peer Environment. *to appear in Concurrency & Computation: Practice and Experience*, 2004.

[4] P.Th. Eugster and R. Guerraoui. Distributed Programming with Typed Events. *To appear in IEEE Software*, 2003.

[5] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):217–246, 1985.

[6] R. Guerraoui, P.Th. Eugster, P. Felber, B. Garbinato, and K. Mazouni. Experiences with Object Group Systems. *Software - Practice and Experience*, 30(12):1375–1404, October 2000.

[7] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[8] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *ACM Transactions on Computer Systems*, 28(9):690–691, September 1979.

[9] B. Meyer. Object-Oriented Software Construction, 2nd edition (chapter "Concurrency, Distribution, Client-Server and the Internet"). Prentice Hall, 1997.

[10] B. Meyer. The Power of Abstraction, Reuse and Simplicity: An Object-Oriented Library for Event-Driven Design. To appear in *Festschrift in Honor of Ole-Johan Dahl*, eds. Olaf Owe et al., Lecture Notes in Computer Science 2635, Springer-Verlag, 2003.

[11] C.H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[12] D. Powell. Group Communications. *Communications of the ACM*, 39(4):50–97, April 1996.