A Benchmark Suite for Distributed Publish/Subscribe Systems

Antonio Carzaniga and Alexander L. Wolf

Software Engineering Research Laboratory
Department of Computer Science
University of Colorado
Boulder, Colorado 80309-0430 USA
{carzanig,alw}@cs.colorado.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-927-02 April 2002

Abstract

Building a distributed publish/subscribe infrastructure amounts to defining a service model (or interface) and providing an implementation for it. A typical distributed implementation is architected as a network of dispatcher components, each one implementing appropriate protocols and algorithms, that collectively realize the chosen service model. The service model should provide a value-added service for a wide variety of applications, while the implementation should gracefully scale up to handle an intense traffic of publications and subscriptions. We believe that the design of such service models and implementations must be guided by a systematic evaluation method, which in turns must be based on a carefully chosen *benchmark suite*. In this paper, we lay out a set of requirements for a benchmark suite for distributed publish/subscribe services, and we outline its primary components. The ideas proposed in this paper are based on our own experience in building and studying publish/subscribe infrastructures, and on existing evaluation frameworks.

1 Evaluating a Distributed Publish/Subscribe Service

Within a publish/subscribe service, information flows from publishers to subscribers according to the specific selection criteria expressed by individual subscribers. Subscribers express their interests by means of subscriptions, while publishers simply publish information. The service accepts subscriptions and publications, and relays publications to subscribers that declared matching subscriptions. The general architecture of a distributed publish/subscribe infrastructure is a network of dispatchers [1, 4, 5, 7]. Publishers and subscribers are directly connected to one dispatcher, to which they send their subscriptions and publications. Every dispatcher processes subscriptions and publications according to some protocol, possibly redistributing subscriptions to other adjacent dispatchers and publications to adjacent dispatchers and/or subscribers.

Realizing a publish/subscribe service according to this general schema amounts to designing

- a *data model* for publishable information (i.e., format, structure, types, etc.)
- the *selection mechanisms* available to subscribers (i.e., scope, syntax, and semantics of subscriptions)
- the *topology* of the network of dispatchers, including the rules by which dispatchers join and leave the network, the protocols used by dispatchers to establish a connection, and possibly to authenticate each other, etc.
- the *communication protocols* used among dispatchers, and between dispatchers and clients. These protocols cover the basic exchange of publications and subscriptions as well as all the auxiliary information flows needed to, say, compute shortest paths and other such specific functions.
- the *internal architecture of dispatchers*, including physical components, such as input/output modules, switching components, and logic components, including matching algorithms, routing algorithms, and their data structures.

Each and every one of these design aspects has been extensively studied, in various forms and combinations, in different contexts, ranging from databases to programming languages, to networking research. The ultimate challenge for the designer of a distributed publish/subscribe service is to engineer existing and new techniques to create a publish/subscribe infrastructure capable of offering an added-value service for the widest variety of applications, and at the same time, capable of scaling from simple localized applications, up to complex, traffic-intensive, highly distributed applications.

Clearly, these goals demand a methodical engineering approach. Specifically, we believe that the numerous degrees of freedom in the design space and the inherent complexity of each aspect of the design emphasize the importance of serious evaluation methods. Even more to the point, we argue that not only methodical performance evaluation and service validation are necessary steps in the design process, but also that they should be considered as the primary guides for the development and integration of technologies for distributed publish/subscribe services.

What we propose as an initial solution to this design problem is the definition of a *benchmark suite*. In the rest of the paper we lay out a set of requirements and some initial specifications for a distributed publish/subscribe benchmark suite.

2 Requirements for a Benchmark Suite

The first requirement for a benchmark suite is in fact a meta-requirement, or rather a condition on the process by which the benchmark is defined. It is crucial that the benchmark suite be widely accepted by researchers and practitioners in the field, and consequently adopted as part of their design environment. It is therefore very important that the formulation of the benchmark itself be a communal activity. In this spirit, we intend this paper as an initial proposal, meant to generate discussion and interest, which we hope would then translate into a cooperative benchmark definition effort.

The specific high-level requirements for the benchmark suite are directly related to the function of the benchmark. As we said, the function of the benchmark can be broken down into two main goals:

- validation of the service interface, and
- performance evaluation.

In order to satisfy the first goal, the benchmark suite must include tests and methods that assess the suitability of a particular publish/subscribe service. Suitability is intended as an aggregate measure of the added value provided by the service to a wide range of applications. Obviously, this evaluation is characterized by qualitative parameters, and is likely to be affected by subjective interpretations. Therefore, the benchmark should guide and disambiguate the evaluation process by establishing a detailed *capability model*.

The second goal is somewhat more concrete, since it is characterized by more objective and quantitative measurements. The benchmark should be based on the cross-product of a number of applications, executed over series of configurations and distributions of their components, under a series of workloads. Performance metrics should be based on time and space complexity. Typical such metrics are end-to-end latency, end-to-end maximal throughput, and maximal footprint for individual components.

Despite its conceptual simplicity, a performance benchmark poses a fundamental obstacle, when combined with a service suitability benchmark. The problem is that an ideal service suitability benchmark should not be biased towards any specific publish/subscribe interface. By contrast, a performance benchmark must consist of concrete application programs, which must use specific service interfaces. One extreme approach to resolve this conflict would be to define a specific interface, as an integral part of the benchmark, and to focus on applications that use that interface. In this case, a designer of a publish/subscribe system with a different service interface would have to provide an adapter that would somehow translate the benchmark service interface to his or her own service interface. The opposite approach would be to build the benchmark out of very service-neutral applications, or better, applications capable of adapting to different service models.

Neither of these approaches seems satisfactory. The second one is probably not feasible at all, since arguably there is no such thing as a real service-neutral application. While the first one poses another dilemma in choosing the type of benchmark interface: a "common denominator" interface would probably be too simplistic, whereas a very rich interface would penalize most services. In any case, using a single interface as a reference would create a restricted and biased benchmark. The natural compromise that we propose is to select applications that use a number of specific publish/subscribe interfaces. The benefit of this idea is that it would create a fair and more generic benchmark. The drawback is that, in order to execute benchmark applications, the service designer would have to provide several adaptation layers.

The choice of applications, configurations, and workloads is clearly crucial for the formulation of the benchmark. We see two distinct requirements for this part of the benchmark suite. On one hand, part of the benchmark must be an expression of real, current applications, independently developed by third-party organizations. Similarly, configurations and workloads must reflect real computing environments and actual use scenarios. Ideally, this part of the benchmark would be derived from recordings of actual application sessions. The objective of this section of the benchmark is to directly benefit current technologies and usage patterns. On the other hand, the benchmark should also account for future application developments and unforeseen usage scenarios. This part of the benchmark suite must rely on synthetic scenarios. These scenarios need not be related to actual usage patterns, and should instead explore situations that are far from the current realm of publish/subscribe applications.

3 Initial Benchmark Specification

Following the requirements we set in Section 2, we propose a benchmark suite composed of three sections: an *interface suitability* section, an *applications* section, and a *synthetic scenarios* section. We will discuss the details of every one of these sections in the following.

3.1 Interface Suitability

The purpose of this section of the benchmark is to establish a *capability model* for publish/subscribe services. The benchmark program is a simple semi-automated self-evaluation system based on questions and answers. The program guides the designer in the evaluation process by asking questions derived from the capability model. It then annotates the designer's answers and eventually computes aggregate evaluation metrics based on coverage criteria expressed in the capability model.

A complete formulation of the capability model is well beyond the scope of this paper, and as we said, it should result from a joint effort of researchers and practitioners in the field. Here we list a number of general features that we believe should be included in the capability model. Elements of this list have been extensively studied within other evaluation frameworks [4, 5, 2, 8].

- *publication model:* data model for publisheable data. This model should classify services according to the following parameters:
  - *structure:* characterizes the structure of notifications. Typical publications can be classified as: unstructured, lists of strings, record-like structures with positional or name-based identification of attributes, recursive structures, such as LISP expressions or XML documents, and composite publications, made of digests of other publications
  - *types:* predefined domains of values. Typical type classifications would be binary or string, simple atomic types (such as integers, dates, booleans), and typed structures, that is, structures whose combination of fields constitute a type in itself
  - *limits:* total byte size, number of attributes, limits for types (string length, integer sizes or ranges of values), and number and depth of sub-structures
- *subscription model:* defines the selection capabilities of the publish/subscribe service:
  - *scope:* defines what parts of a publication can be evaluated and selected within subscriptions. A typical classification of scope may be the following: single globally known field, single subscription-specific field, limited number of fileds, entire publications, and groups of correlated publications
  - *language power:* characterizes the language that defines subscription in terms of its expressive power. Typical language power classifications are: simple predicates, such as equality, inequality, and string operators, and composite expressions
  - *language style:* declarative, imperative
  - *other features:* extensibility (for example, by means of plug-ins), useful special operators such as a "certificate-based authentication" predicate that would select all the publication that a client can successfully authenticate.
- *interface access methods:* support for local access, for example, with shared memory or other OS in-memory connections such as UNIX sockets or signals, and remote access, with specific communication protocols
- *portability:* support for multiple platforms and language bindings
- *service model:* examples are reliable/unreliable for point-to-point communications, end-to-end reliable/unreliable, quality of service negotiation, persistent store-and-forward service
- *auxiliary features:* for example, security and support for mobility

3.2 Applications

This section of the benchmark suite is intended to evaluate the performances of the publish/subscribe infrastructure, when serving current common applications, under observed traffic loads. For these benchmarks, we propose to consider applications within the following categories:

- *internet-based trading:* this includes auction systems (such as eBay), airline reservation systems, and other generic e-commerce applications. This class of applications is characterized by high volumes of traffic, with sparse subscribers over a wide-area network. Subscribers are likely to select very specific information out of a varied information space

- *news:* network-based sportscast systems, financial news systems, emergency announcements, and general event notification. Clearly, this is also a class of applications that span wide-area networks. However, in contrast with internet applications, it is likely to have more dense distributions of subscribers, with more generic and/or less volatile requests, with a higher traffic flow
- *networked interactive games:* this class of applications is interesting for its essential real-time requirements, as well as for the very intense, but probably localized, traffic
- *software systems:* this category includes workflow management systems, application management, distributed agenda management, and mobile agents platforms. Application from this category are likely to require the exchange of complex objects, and therefore are likely to exploit the most advanced data modeling and selection features of the publish/subscribe service.

These benchmarks also define the precise workload and the precise component configurations, in combination with each individual application. We propose to define these workloads and configurations by recording several actual sessions for each application. We realize that this may be a considerable effort in itself, and that the measurement process is likely to pose serious software engineering and measurement challanges.

3.3 Synthetic Scenarios

Synthetic benchmarks are meant to test a publish/subscribe service under uncommon workloads, or under traffic patterns and configurations not exhibited by current applications. Synthetic benchmarks should also focus on specific features or traffic patterns, since this can not be easily done with workloads derived from actual applications. The use of synthetic workload generators is common practice in the evaluation of network protocols and network architectures. Synthetic scenarios have also already been used to evaluate a number of techniques for publish/subscribe services [4, 6, 3].

Being actual programs, synthetic benchmarks too are designed to use a specific service interface. Therefore, not every benchmark program may be applicable to each and every service interface. This section of the benchmark suite must contain enough benchmark programs to cover the most common service interfaces. In the following, we will present an example of a synthetic benhcmark that uses a specific publish/subscribe service model, nontheless the basic ideas expressed by the example are valid for other service models as well.

The benchmark is essentially composed of two parts:

- *topology:* this part defines the distribution of service components and application components over a network. The topology is expressed in terms of the connections between components, and their costs.
- *application behavior:* this part defines the combination of service requests, including specific values for publications and subscriptions.

The topology may be defined directly at the application level or as an overlay of network and application topologies. The substrate network topology can be generated using an appropriate random graph model [9]. It is unclear to us whether the same type of models can be adapted to generate plausible application-level topologies (we were unable to find a relevant study of application-level topologies in the literature). If the substrate is a network-level topology, an application interconnection must be set up on top of the substrate network. This can be done in a number of ways. What we propose is an incremental process, which works as follows: the number of application-level nodes is given as an initial parameter. The first "root" application-level component is allocated on a randomly chosen node of the substrate. The remaining application-level nodes are randomly placed and randomly connected to a number of existing components. The way these two random choices are made depends in part on the type of service architecture, and in part on other benchmark topology parameters. For example, if the service requires an acyclic architecture, then new nodes will be connected to exactly one existing node. The remaining random parameters can be adjusted by using distributions that favor locality in both placement and interconnections. Once the service topology is defined, the benchmark must allocate application components on each host.

This process can follow the same schema described above, again maintaining some reasonable form of locality in connecting applications to service endpoints.

The next step in the formulation of this sample synthetic benchmark consists in assigning behaviors to application components. This part of the benchmark is characterized by several degrees of freedom, and therefore lots of parameters. What we give here is a list of parameters that subsumes a generic randomized behavior. Specifically, we assume periodic event generators, as well as periodic subscribers. That is, publishers that emit publications at a fixed rate, and subscribers that continuously subscribe (for some information), receive a number of publications, unsubscribe, wait some time, and then re-subscribe. These are the essential parameters:

- *publication rate:* number of publications issued per time unit (poisson distribution)
- *active subscription cycle duration:* interval during which a subscriber has at least one active subscription. It may be defined by the number of received publications, or by a simple timeout
- *inactive subscription cycle duration:* interval in which a subscriber has no active subscriptions. This is a simple interval between unsubscriptions and subscriptions in a subscriber's cycle
- *publication model:* parameters used to generate publications (remember that, in this example, publications are record-like structures):
  - *number of fields:* this should also be randomly determined according to a Poisson distribution (since the value can never be less than one)
  - *namespace for attributes:* set of names used to identify attributes in notifications. We propose to use a random selection out of a dictionary of words (e.g., a standard English dictionary)
  - *prevalence of attribute names:* probability function for attribute names within the namespace
  - *prevalence of types:* probability functions for attribute types
  - *values space:* set of values, one set per type. This parameter defines ranges for numeric values, and dictionaries for string values
  - *prevalence of values:* probability function for values
- *subscription model:* parameters used to generate subscriptions, and to associate them with subscribers. In this example, subscriptions are conjunctions of elementary predicates, each one defined over the value of a given attribute. Predicates, like attributes, are typed, meaning that they match attributes of a specific type:
  - *number of subscriptions per subscriber:* a subscriber may have several active subscriptions at the same time. This parameter determines the distribution of subscriptions over subscribers
  - *number of predicates per subscription:* randomly distributed
  - *name space for subscriptions:* set of attribute names used in subscription predicates. This set is composed of words extracted randomly from a dictionary and from the namespace for publications
  - *percentage of shared namespace:* indicates what fraction of the namespace of subscriptions is extracted from the publications namespace. This parameter roughly measures the degree of integration between publishers and subscribers
  - *prevalence of attribute names:* probability function for names in subscription predicates
  - *prevalence of types:* probability function for types in predicates
  - *prevalence of predicate operators* probability function for operators (equality, greater-than, lower-than, etc.)

4 Conclusions

We are convinced that systematic performance evaluation and service validation are necessary, crucial steps in the design of publish/subscribe infrastructures, especially those based on distributed architectures. With this paper, we propose the formulation of a benchmark suite as a basis for that evaluation effort. We also believe that the definition of the benchmark suite must be a joint effort of researchers and practitioners in the field of publish/subscribe systems. Therefore, we intend our proposal as a stimulus for discussion, and possibly a starting point for the formulation of a comprehensive and widely accepted benchmark.

Acknowledgments

The work of the authors was supported in part by the Defense Advanced Research Projects Agency, Air Force Research Laboratory, Space and Naval Warfare System Center, and Army Research Office under agreement numbers F30602-01-1-0503, F30602-00-2-0608, N66001-00-1-8945, and DAAD19-01-1-0484. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory, Space and Naval Warfare System Center, Army Research Office, or the U.S. Government.

References

[1] G. Banavar, T. D. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *The 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, pages 262–272, Austin, TX, May 1999.

[2] D. J. Barrett, L. A. Clarke, P. L. Tarr, and A. E. Wise. A framework for event-based software integration. *ACM Transactions on Software Engineering and Methodology*, 5(4):378–421, Oct. 1996.

[3] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *Proceedings of the 23th International Conference on Software Engineering*, pages 443–452, Toronto, Canada, May 2001.

[4] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug. 2001.

[5] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 9(27):827–850, Sept. 2001.

[6] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *ACM SIGMOD 2001*, pages 115–126, Santa Barbara, CA, May 2001.

[7] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Struman. Exploiting IP multicast in content-based publish-subscribe systems. In J. Sventek and G. Coulson, editors, *Middleware 2000*, number 1795 in LNCS, pages 185–207, New York, NY, Apr. 2000. Springer.

[8] D. S. Rosenblum and A. L. Wolf. A design framework for Internet-scale event observation and notification. In *Proceedings of the Sixth European Software Engineering Conference*, number 1301 in Lecture Notes in Computer Science, pages 344–360. Springer–Verlag, 1997.

[9] E. W. Zegura, K. Calvert, and M. J. Donahoo. A quantitative comparison of graph-based models for internet topology. *IEEE/ACM Transactions on Networking*, 5(6), Dec. 1997.