



BENCHMARKING ENTERPRISE MESSAGING SYSTEMS

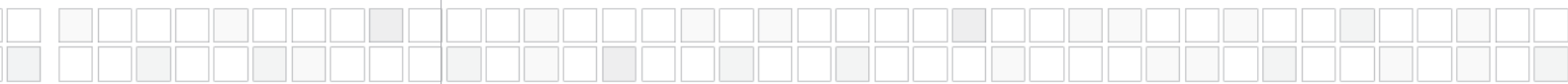


TABLE OF CONTENTS

Abstract	2
> 1.0 Introduction	3
1.1 Messaging: The Backbone of the Enterprise Service Bus	4
1.2 The Importance of Messaging Platforms	4
1.3 Defining the Purpose of the Test	5
> 2.0 Characterizing Requirements	5
2.1 The Integration Architecture	6
2.2 Endpoints	8
2.3 Services and Processes	9
2.4 Clients	11
2.5 Asynchronous versus Transactional	11
2.6 Message Data Model	12
2.7 Endurance	13
2.8 Scalability	14
2.9 Hardware Platforms	15
2.10 The Benchmark Requirements Document	16
> 3.0 Specification of the Benchmark	17
3.1 Defining Prototype Processes	18
3.2 Isolating Enterprise Messaging Factors	18
3.3 Characterizing the Messaging Backbone	19
3.4 Simulating External Factors	19
3.5 Defining Input Load	20
3.6 Hardware Options	20
3.7 Making the Specification Realistic	21



> 4.0 Building the Benchmark	22
4.1 Independent Variables	22
4.2 Dependent Variables (Measures)	23
4.3 Control Variables (Constants)	23
4.4 Configuring Services and Endpoints	24
4.5 Driving the Test with the Sonic Test Harness	25
4.6 Configuring the Test	27
4.7 Customizing the Test Harness	30
4.8 Custom Service Simulations	31
> 5.0 Executing the Benchmark	32
5.1 The Performance Test Cycle	32
5.2 Configuring and Running Tests	32
5.3 Collecting and Evaluating Results	34
5.4 Tuning and Tweaking	36
5.5 Analyzing the Cost/Benefit of Tradeoffs	37
> 6.0 Performance Engineering and Cost/Benefit	38
6.1 Progress' Commitment to Scalable Solutions	39
> 7.0 Conclusions	40
> About the Author	40



ABSTRACT

This white paper is written for integration architects involved in planning and deploying messaging-based systems. It discusses the role of performance analysis in this environment and then gets into details on how to plan and execute an appropriate performance benchmark to answer key questions. Successful benchmarks depend first upon accurate performance requirements definition, based on current and future realities of usage, load and scalability. Once this is understood, we discuss techniques to translate these requirements into a concrete benchmark specification that can guide the actual testing. The next step discussed is implementation of the benchmark with an emphasis on making the test an accurate simulation of the original performance requirements. After this, we cover the iterative stage of benchmark execution, including techniques to improve the quality of information returned while reducing time and effort. Finally, the paper concludes with a discussion of how to interpret the benchmark results in the larger context of the total cost/benefit of the solution.

The discussion includes a description of the Progress® Sonic™ Test Harness, an open source tool for JMS/ESB benchmarking available on the Sonic Web site. It does not go into detail concerning Web service testing since that topic is sufficiently covered elsewhere. The paper focuses on design, high-level concepts and techniques; for details on tools and performance measures, consult the user documentation for the Sonic Test Harness.



1.0 INTRODUCTION

A key consideration for enterprise integration is the risk and cost associated with system performance. Unfortunately, the runtime performance and bottlenecks of a future production system are not easily simulated in the lab. On the other hand, carefully planned benchmark testing should be able to validate whether the proposed solution will scale and perform appropriately, given hardware plans. This information can substantially reduce operational risk or, at least, provide early warning that alternative solutions may be required.

While it is highly desirable to execute benchmark tests for the proposed integration solution, implementation of the benchmark can be very complex and requires careful planning to ensure success. This paper will discuss how to plan and conduct a comprehensive benchmark for integration solutions based on enterprise messaging solutions, including Web services and Java Message Service (JMS) implementations. This kind of asynchronous, loosely coupled integration is exemplified in the enterprise service bus (ESB), an integration technology that combines enterprise messaging, Web services, lightweight deployment containers and distributed management to achieve near real-time enterprise integration. We will focus on the issues involved in accurately measuring performance and scalability for developers who are comparing performance of multiple alternatives. Please visit <http://www.progress.com/sonic> if you are interested in other aspects of message-based integration or if you need additional information on this subject.

Progress solution engineers provide services to Sonic customers who are analyzing ESB and JMS performance for configuration planning as part of a review of design alternatives or to evaluate alternative solutions. They have used the open source performance test harness described below in a number of consulting engagements for performance analysis. This white paper shares the experience, guidelines and best practices developed by this team, which we offer for informational purposes to anyone using Progress® Sonic ESB® or any other enterprise messaging-based infrastructure. It will provide a roadmap for planning and executing your performance evaluation project.



1.1 MESSAGING, THE BACKBONE OF THE ENTERPRISE SERVICE BUS

The enterprise service bus (ESB) is a lightweight framework that provides flexible, loosely coupled integration across a broad range of enterprise applications. The key to dynamic interconnection across security firewalls, application protocols and languages is the enterprise messaging backbone. This component provides guaranteed delivery of messages among services and endpoints, performing security checks and protocol conversions on the fly. Advanced ESBs use a messaging system that ensures fault tolerance with near real-time failover in case of system faults. The messaging system is the nervous system of the ESB and is, hence, central to evaluating the current and projected performance of a deployed ESB. The messaging system is usually also the on-ramp for business processes, and, as such, it dictates the content and rate for business requests entering the system.

1.2 THE IMPORTANCE OF MESSAGING PERFORMANCE

Clearly, it is difficult to know in advance how aggregate message loads will evolve in an ESB. Depending on the type of service, it may be necessary to service thousands of requests per second through a single messaging endpoint. For request-reply (synchronous) services, sub-second message latency is often necessary to meet corporate quality of service (QoS) goals. The integration team is normally tasked with defining the hardware requirements to meet current needs and sometimes with projecting requirements into the future.

The ESB ensures unprecedented scalability in the enterprise by allowing services to be replicated across dozens, or hundreds, of lightweight service containers. Using the standard publish/subscribe mechanisms of the JMS infrastructure, the stream of requests is load-balanced across service instances, providing parallel execution of business processes. The ESB also supports less robust protocols, including SOAP over HTTP, which permit Web service access without modification of existing infrastructure. The scalability of the ESB depends on the ability to manage arbitrarily increasing message loads. Consequently, it is important that a single messaging server be able to manage high rates of traffic, but even more essential that the messaging mechanism itself provides for extended scalability, based on multithreading, connection pooling, broker clustering and routing mechanisms.

1.3 DEFINING THE PURPOSE OF THE TEST

Before you make the considerable investment of running a performance benchmark, you should be clear on the purpose of the testing. This may be one or more of the following:

1. Planning future system capacity needs
2. Comparing the efficiency of various integration design options
3. Comparing the cost/benefit of enterprise messaging providers

This purpose will guide the decisions and priorities of the performance evaluation project and will help keep the testing and analysis focused as the project proceeds.


Equally important is to ensure that the tests you run are a realistic assessment of the value of each alternative in terms of the performance, scalability and costs you can actually expect to encounter. In our years of benchmarking and analysis of messaging systems, we find that there are several areas that are particularly important in ensuring the value of performance tests:

- > Mimicking current or future system loads
- > Simulating the topology of the messaging infrastructure, including endpoints (defined below), routing paths and services
- > Avoiding artificial or unnecessary constraints on the potential implementation
- > Prioritizing those components and requirements that have a high impact on performance

The final observation is that there is no substitute for experience. If you can involve professionals who have done performance testing and tuning before, it will improve the value and relevance of your results. The remainder of this document will discuss planning, building and executing a performance benchmark to meet these goals.

2.0 CHARACTERIZING REQUIREMENTS

Since a performance load test is a significant investment, you'll want to carefully analyze the requirements that identified the need for testing. This analysis includes some brainstorming to determine those aspects of your solution that are already fixed, versus those decisions yet to be made, versus those that are firm but require some validation.



The ideal time to begin the benchmark design is about halfway through the design phase of the integration project, so you have some structure to build on, but haven't fully committed to untested or risky alternatives yet. We advise drafting a written document that outlines the unusual or risky aspects of the project (which may include introduction of an ESB) and describes how benchmark testing will provide the facts needed to make tough choices and reduce overall project risk. This **benchmark plan** will help justify the resources required for the benchmark and provide a basis for a full project plan including milestones and measurables. You will find your chief technology officer (CTO) is very receptive to such a plan, as it helps him or her identify and manage real risks in the project.

You can begin the analysis by reviewing the current design for the integration architecture and focusing on those aspects of the architecture that may be subject to high runtime load. Your goal is to identify those **services, endpoints** and **processes** for which improved performance and load management would provide substantial benefits. For endpoints and services, there is likely to be a short list of questions and design alternatives you can address with selective testing. For distributed service-oriented architecture (SOA) processes, you need to do a more complex analysis of design and deployment options. This includes the use of asynchronous versus transactional interfaces, routing options and the use of XML. This section summarizes some of the concepts and details we've found to be important to the benchmark planning process.

2.1 THE INTEGRATION ARCHITECTURE

The most common messaging benchmark is to have a single message producer send messages via a single JMS destination to a single consumer, commonly known as the "1-1-1 scenario." This is also, perhaps, one of the least likely cases to occur in a deployed ESB. The test can be refined by putting the producer, broker and consumer on different machines and increasing the number of concurrent sessions on each side. It may be the case that the only subset of the ESB that has a sufficiently high load to be of concern is, in fact, a simple single-hop message transfer. Since you are risking the validity of the entire benchmark project on the assumptions you make here, it is worth taking extra time to analyze the proposed architecture and validate that the test specification includes all relevant aspects of message distribution, fan-out and interaction. Restrict this analysis to the use cases that have scalability or capacity considerations, or else the scope may expand to unmanageable proportions.

You should also be cautious not to over-specify the benchmark test. For example, new messaging destinations can be implemented using either point-to-point or publish/subscribe domains. Unless there is a driving technical requirement for either of

these, you should leave the test specification flexible so that the benchmark implementation can focus on whichever is more efficient (see Endpoints and Services below). Another important design area is the management of data integrity. There are different ways of ensuring data integrity, including messaging reliability options (see Message Delivery below), JMS transactions, database transactions and logging services. If performance and scalability are a high priority, you may want to allow some flexibility here and try more than one option. Keep in mind that any performance benchmark is a learning experience, and you are very likely to gain insight into the design alternatives for your architecture if you allow selective experimentation in key areas.

We recommend that you begin the benchmark planning phase with an understandable diagram of the architecture. Some of the elements that should be included in that diagram are shown in Figure 1.

Figure 1. Elements of the Integration Architecture Topology

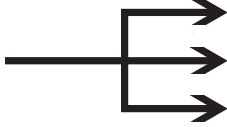
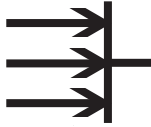

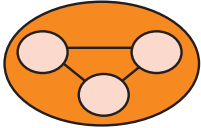

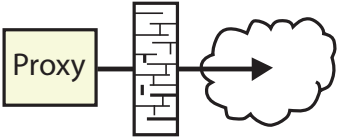
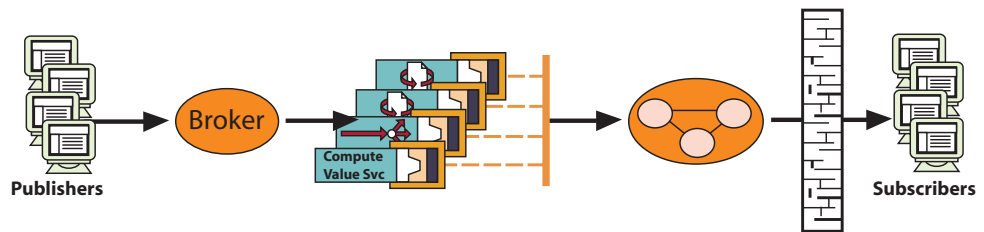
Fan-out		A single message or event generates multiple message sends: for example, a Topic endpoint with multiple subscribers or a batch message split into individual events.
Fan-in		Several messages or events are joined together and consumed within a single service; this may include services that perform multiple Web service callouts, an orchestrated "join" or a consumer that batches transactions.
Routing		Decision points within a process that selects among several different endpoints or services.
Brokers		Allowable configuration of broker clusters and dynamic routing (i.e., store-and-forward paths).
Transformation		Service invocations that perform data transformation operations.
Network		Layout and specification of the network topology, especially firewalls, LAN connections, etc.

Figure 2 illustrates a sample topology for an ESB test. Note that the test sequence could include subsets of the overall topology. For example, you could simulate the load through the initial broker in one test and the second hop (through the broker cluster) in another test and then put them both together with the ESB services. You don't need a very detailed or fancy architecture diagram; you just need to communicate the pattern of message transfer and consumption, so that all participants have an agreed roadmap for determining which tests should be done and how each relates to the proposed solution.

Figure 2. Example Test Diagram




2.2 ENDPOINTS

Real-world ESBs provide a range of design alternatives for distributing messages. For instance, they allow a single, high-volume message source to be split into several smaller, more manageable streams. The ideal enterprise messaging test specifies the message traffic in terms of abstract endpoints and allows broad flexibility in determining how those endpoints are deployed. In that way, a specific vendor tool or design approach can be chosen that will provide a scalable implementation without compromising the architecture or application portability.

The specification for endpoints will determine the **protocol** and **messaging domain** used in delivery of the messages. The default protocol for the ESB is JMS over TCP/IP. But you can also configure SSL, HTTP or HTTPS, and any of these can have the SOAP Web services protocol layered over it. The ESB should also support on-ramps using FTP, SMTP (mail) or even file pickup and drop. For JMS-based messaging, you have the choice of publish/subscribe (PubSub) or point-to-point (PtP) domains. PubSub is generally more flexible for dynamic message routing, but PtP allows more administrative control over a queue. For Web services, you can deploy as HTTP Direct, SOAP/HTTP, SOAP/JMS or WS-Security. You have the option of testing any combination of endpoint protocols, and the good news is that you can reconfigure any of your endpoints without causing any impact on the deployed services, processes and containers.

A key aspect of endpoint specification is defining the parameters for message delivery, including **quality of service** (QoS), security, transactions, retry and logging. Within the



JMS specification, the message producer has options that define the persistence of the message, which is, in effect, a contract with the broker to guarantee delivery regardless of broker failure. Also related to QoS is the specification for **acknowledgement mode** on the message consumer's side, which determines when the broker can safely assume the message is delivered and can, therefore, release any broker resources or wait conditions on the sender associated with the message. JMS transactions are an important way of ensuring that message delivery is properly synchronized with related application actions or will be retried in the event either fails. Transactions add additional latency and/or logging overhead, especially if they conform to the XA transaction specification. Much message logging is transient, internal to the broker's guaranteed messaging algorithms, but some enterprise messaging systems provide for permanent logging of messages for audit, replay or business activity monitoring (BAM) purposes. Always remember, though, that the cost of disk logging is often an order of magnitude greater than the cost of network message delivery. Sonic ESB provides real value here, by supporting **non-persistent replicated (NPR)** mode. This mode uses the real-time messaging replication mechanism of Sonic's Continuous Availability Architecture (CAA) to implement QoS based on cross-broker replication, rather than disk log writing. Since network-based operations are substantially faster than disk, NPR mode can improve performance substantially without sacrificing message guarantees.

2.3 SERVICES AND PROCESSES

The performance of business processes that have multiple steps can be very sensitive to the latency of any service invocation in the chain. The reason is that the messaging subsystem may be required to retain or log messages pending acknowledgement from the service in order to ensure the requisite quality of service for messages (see the following section). In a performance benchmark, it is tempting to represent services with simple stub programs for simplicity. These can save time, but some attempt should be made to either simulate the time and resource required by the service or to validate that the impact will be inconsequential. On the other hand, it is important to use the actual implementation for services that are tightly coupled with the messaging/service infrastructure, including **transformation, content-base routing, logging or file/FTP** services.

In the ESB, services run within lightweight Java processes called **service containers**. Each service container is deployed on a specific host and connected to the messaging backbone. You deploy each service in any number of these service containers, specifying how many execution threads to allow. Therefore, your test specification should address the mapping of services into containers, the number of containers deployed across test hosts and other tuning/sizing information for the container. You can attempt to define



these upfront or, as is more typical, make it a goal of the testing to develop a recommendation in this area. As an alternative, service logic can be deployed as EJBs, stand-alone Web services or Web servlets with similar specification requirements.

An ESB **process** is a set of sequential or branching service invocations that performs a meaningful business transaction. Indeed the whole point of an ESB is to take a set of disparate services and allow a single process to engage several of them, managing all necessary transformation and routing within the bus, so you can leverage each service as a shared corporate resource without having to modify it. A key part of the benchmark specification will be to identify the complexity of ESB processes and the mechanisms used to implement them. Progress Software is the only vendor that supports all three ESB process implementation mechanisms:

1. Channel based: i.e., connecting the output destination of one service to the input destination of another
2. Itinerary based: i.e., attaching process state to the actual message event, so that process steps can be executed in any container
3. Persistent process: i.e., storing the process state locally to a BPEL process engine

These are presented in order of increasing flexibility but decreasing performance, so your test plan should give some consideration to how you will analyze the tradeoffs between performance and capability inherent in choosing process implementation type. For instance, it is a common design decision to implement high-load **logging** patterns using channel-based routing, moderate-load **batch file** patterns using itineraries, and light-load, **split join** patterns with persistent processes. In addition to the process mechanism, you should analyze the most critical processes (i.e., the high-load or high-risk ones) to project the number of steps and pattern of **splits** that are expected. If you use persistent process engines (such as Progress® Sonic™ BPEL Server™) you need to evaluate the process engine's distribution and resource requirements the same as you would for any other service.

2.4 CLIENTS


Test clients can represent the on-ramps of the ESB, i.e., any message producer or consumer outside the staged ESB being tested. These will be JMS or HTTP(S) clients and are generally simulated in the benchmark using a test tool such as Sonic Test Harness. The key issue in specifying test clients is to determine the mapping of clients into the endpoints and the load range for each. Other key client characteristics may be specified as variables, so that you can determine the range of operation in order to guide further refinement of the topology. This especially applies to the number of connections, sessions and threads used for messaging, and these can have enormous impact on the number of messages each client can handle. The mapping of sessions to endpoints (which typically include a named topic or queue) is also important. Of course, you need to restrict the session/endpoint mapping to options that would be realistic if implemented in production. Figure 3 shows an example of a specification for session/destination mapping.

Figure 3. Specification of Client/Destination Mapping

Driver A	Driver B
"Event publisher"	"End user app"
1 connection	100 to 200 connections
1 to 20 sessions per connection	1 session per connection
Publish to 10 topics total	Subscribe to 4 topics at random
Message rate total 1500 msg/sec	Message rate max achievable

2.5 ASYNCHRONOUS VERSUS TRANSACTIONAL

Application servers were originally built to manage synchronous transactions in which the inbound request is managed by a unique session within the server. During execution, the session ties up limited resources, including threads, memory structures and network connections. Perhaps most important, the approach assumes the client itself is suspended, pending receipt of the response. This **request reply** model requires management of session state across client, messaging brokers, service containers and ESB processes, increasing the resources consumed. Multithreaded clients allow some overlap in processing, but, as loads increase, it is typical that service latencies will also increase, leading to an exponential increase in the number of threads required to service the load. In order to ensure sub-second response time, application server designers



typically either upgrade to expensive, high-end hardware or compromise the functionality of the application. As business processes become more complex and more difficult to micro-tune, however, scalability becomes problematic. Thus the benefits of the **transactional** approach in ensuring data integrity are gained at the cost of a negative tradeoff in throughput, scalability and system resilience.

With an ESB, the asynchronous approach is preferred, meaning that the client program does not wait for a transaction commit. Instead, the messaging infrastructure uses sophisticated acknowledgement, replication and logging techniques to provide a guarantee that the message will be properly delivered. Confirmation or rejection of the business request can be routed back to the user in a reply-to queue after all steps in the business process have been executed. This approach allows resources to be freed for other work, enables integration of non-transactional (batch) business applications, and more easily accommodates variations in peak loads and availability. It is still possible to use transactions in steps or sub-processes at those points where strict data serializability is required.

2.6 MESSAGE DATA MODEL

Enterprise integration via the ESB ensures loosely coupled, dynamic interaction across service interfaces, but this interaction requires that the structure and content of the data being passed back and forth are appropriate and correct. Traditionally, this integration is done using a **static metadata** approach in which some central data structure definition (usually an SQL or XML schema) is used to define and validate the structure of shared and stored data. The ESB uses a much less restrictive approach that requires instead that any transition from one service to another requires a defined **transformation** to convert the data and a well-defined routing for any validation errors. In the ideal case, in which both applications can agree on the static schema, this step is skipped, but in other cases the transformation step is configured to automatically convert data within the process. Because this step can be a considerable consumer of CPU resources, you'll want to include a realistic transformation in your benchmark test processes.

These **transformation** and **content-based routing (CBR)** services are key capabilities of the ESB and have a dramatic impact on both the message load and the latency of a business process. There are several solutions available with Sonic ESB for message data model:

-
- > XML-based transformation and routing via built-in ESB services
 - > Text-based transformation and routing via custom ESB services and/or ETL products embedded as ESB services
 - > Full semantic rules integration via Progress® DataXtend™ Semantic Integrator

In all cases, the performance of these services is closely tied to the actual content of the messages. For estimates of transformation latency to be meaningful, for example, the test must use messages of similar size and complexity. When messages are distributed among endpoints using content-based routing, it is highly desirable that the test environment generate test messages that will split across the routing destinations in about the same proportions that are expected in the production environment.

Note that it is much less important for the messages to have realistic data values than it is for those values to behave in the expected way. For example, it is seldom important to specify the content of general text fields other than to define the variation in size, but it can be very important to assign the values of key fields used in content-based routing rules.

2.7 ENDURANCE

Most ESB tests reach a steady state in less than a minute, but there are situations and products that produce eventual degradation under load. For this reason it is advisable to run at least a few of the tests for extended periods, hours or even days, to make sure your solution will maintain its robustness and performance over time. There are two areas where extended duration tests can identify issues:

- > When there are unreclaimed resources, such as a memory leak or orphaned system threads, the extended test will show resource usage constantly growing.
- > If there is potential for hidden conflicts, such as a resource/database deadlock or thread contention, the extended test is more likely to expose the problem.
- > If any sub-components, such as non-indexed database retrievals, do not scale linearly, the extended test will show the trend.

If none of these apply, our usual default for individual benchmark iterations is to run ten 10-second intervals (the total 100-second test time makes time rate computations very simple). If there are endurance concerns, you will want to schedule very long-running tests so you can complete them in the time allocated for the benchmark. Keep in mind



that things sometimes go wrong in a benchmark, and you also will find the opportunity to revise and rerun as your understanding grows. Consequently, it is not wise to create a schedule in which every test must go as originally planned in order to complete the project on time.

2.8 SCALABILITY

There is no single way to characterize, or achieve, scalability: it is the sum of performance measures and techniques that come into play as you increase demand. While in theory you should be able to solve any performance problem by expanding the hardware capability, there are always limits to the "linear" scalability curve. We find that ESB scalability depends, to a great extent, on the architecture of the ESB, including:

- > Implementation of guaranteed message delivery
- > Management of message congestion (i.e., what happens when producers get ahead of consumers)
- > Transformation engine
- > Routing engine(s)
- > Clustering and load-balancing options
- > Persistence of business processes


Specification of target scalability can become very complex, and the model you use to define goals can become obsolete as the benchmark proceeds and your knowledge grows. As a result, rather than trying to define precise scalability goals for each architectural component in the benchmark plan, we advise that you focus instead on defining the required load levels at the on-ramps and off-ramps of the ESB. You then manipulate the resources available (e.g., varying the number of service threads, ESB containers or subtopics used) and specify an approach to aggregate the outcomes of these multiple tests. Finally, you plan to analyze these test results as they apply to the original throughput goals with the hope that you will find multiple acceptable solutions. Then you can choose among these alternatives based on the overall business benefits.

2.9 HARDWARE PLATFORMS

There can be substantial and somewhat unpredictable differences in performance between operating systems and machines with different numbers of CPUs. You should do whatever you can to make the test hardware simulate the expected production hardware. If one of the test goals is to choose hardware, you need a broad enough array of test machine options so that you can differentiate among them. In some cases another department, or one of your vendors, can provide access to test farms that can help get definitive answers for you. The next step is to make an attempt to measure the **baseline performance** of individual hardware components. The hardware performance measures most relevant to the messaging system benchmark are:

- > **CPU capacity**—This normally correlates with the number of CPU chips or cores multiplied by the clock speed of each (i.e., two dual-core CPUs at 3 gigahertz would give a total of 12 gigahertz CPU power), but more sophisticated measures of computer speed can also be important. Typically, you also want to measure memory access speed, and the combination of these two will provide a guide for scaling the number of primitive in-memory operations that can be performed by the message broker or service container in a finite amount of time.
- > **Disk speed**—Disk I/O overhead comes into play when a messaging broker is using a disk log to achieve fault tolerance (note the discussion of **non-persistent replicated** mode, above) or when a service invokes file or DBMS operations. Since reliable disk writes depend on the primitive **disk sync** operation, the most useful measure of this capacity is the maximum disk syncs per second. This can be measured by the DiskPerf program, which is bundled with the Sonic Test Harness.
- > **Network speed**—The network performance for a given test is experienced as both **latency** of message delivery, which contributes directly to response time, and **bandwidth**, which limits maximum message throughput. The network path may include network interface cards (typically 100 megabit/second or 1 gigabit/second), LAN switches, WANs and firewalls. You measure network performance using standard **ping** and **traceroute** utilities or the NetPerf program bundled with the Sonic Test Harness.

In our experience, it is more important to benchmark with machines that are of comparable speed to the planned production environment than it is to have the correct number of machines. If you benchmark on lower powered systems, you can roughly extrapolate out to systems about two times faster, but it is dangerous to assume performance will continue linearly much beyond that. If you test on a handful of machines, on the other hand, it is often possible to determine to your satisfaction that



your ESB design can scale to hundreds or even thousands of systems without encountering a performance bottleneck. The reason is that the distributed nature of the ESB ensures there are no "central" resources involved and that load balancing can effectively distribute load across many instances of the same service.

2.10 THE BENCHMARK REQUIREMENTS DOCUMENT

At this point in the project, it is desirable to formalize the above information in a requirements document that can be reviewed by all the participants. There are two reasons it is important to put requirements in writing. First, an organized requirements document ensures that misunderstandings concerning priorities, plans and potential options can be identified early, preventing wasted effort going down "dead ends." Second, explicitly identifying the connection between the tests and the business benefits that can be achieved helps you gain the managerial support and organizational resources needed for a successful project.

There is no need to produce an elaborate requirements document, as most details will be provided in the benchmark specification document that follows. Instead, the document should focus on those aspects of the integration project that involve risks and benefits to the business and the scope of the performance testing project. The document is normally a collaborative work of business requirements experts and integration architects. It communicates the business purpose of the performance test to the IT stakeholders and the core **performance scenarios** to the performance test team.

Based on our experience, we suggest the following outline for this document:

- > **Overview**—a concise description of the reason for the performance project and how it fits into the larger software engineering effort
- > **Scope**—the high-level goals, decision-making process and deployment environment
- > **Scenarios**—a handful of generic integration scenarios that represent the key business processes impacted by performance. For each scenario include:
 - > **Scenario process**—the generic pattern of service interaction for the group of business processes represented by this scenario
 - > **Operational constraints**—the security, quality of service and fault tolerance rules applying to these processes

> **Performance criteria**—the goals for throughput and response time for the business processes as a whole

> **Business value**—the risks and benefits of achieving different levels of performance for this scenario


3.0 SPECIFICATION OF THE BENCHMARK

It is essential to have some kind of specification that defines the purpose and boundaries of the benchmark, although the length and formality of the specification will vary considerably. It is important that the specification provide a concrete proposal to test the requirements and scenarios described in the requirements document. The specification must define all the factors that are necessary to answer the questions raised by the performance project goals, but not unnecessarily constrain the test design and procedures. A careful balance of these two objectives will improve the quality of information returned by the project.

The benchmark specification will drive the benefits and costs for the project, so it is worth spending some time to get it right. Based on the experience in the Sonic performance labs, we offer the following advice in building the benchmark specification:

1. Think first: carefully analyze the benefits, costs and risks for the integration project and prioritize benchmark tests accordingly.
2. Be lazy: leverage benchmark tools to reduce mundane work, and try to answer each question with the simplest possible test.
3. Build bottom-up: test individual components first, and use their performance as a baseline as you combine into larger processes.
4. Iterate: learn as you go and revise the test plan accordingly.
5. Eschew¹ extrapolation: it is generally true that extrapolating results outside the range of inputs tested is highly unreliable.
6. Question assumptions: the primary objective is to determine the best integration solution, so be open-minded about alternative approaches.
7. Respect ignorance: understand those factors that you don't know, and build some learning exercises into the plan.

¹ Webster's Unabridged Dictionary: es-chew', v.t.; to avoid; to shun; to stay away from.



The benchmark specification is the key to ensuring that your benchmark numbers map to your actual ESB solution; without it your results may be meaningless. This section describes the steps we have found useful in defining a realistic benchmark specification that can be implemented within a scoped timeframe.


3.1 DEFINING PROTOTYPE PROCESSES

At this point you should use ESB design tools to create a working prototype of the most performance-critical business processes for the integration project. Since the focus of the project is performance and load balancing, you can simplify the prototype relative to the full business process specification. For example, branches of the process that manage exceptions or rejected messages are very important in functional testing, but probably not a priority for performance testing. On the other hand, services and endpoints on the critical path need to be accurately simulated, especially transformation and routing services.

3.2 ISOLATING ENTERPRISE MESSAGING FACTORS

These factors include the application requirements that bear on the production, dispatch, routing and transformation of messages within the ESB or other integration framework. Focus only on those applications on or off the ESB that contribute substantially to the overall message load, and specify only the basic characteristics about how messages are managed, leaving out details of application logic. You especially need to understand the routing patterns across the enterprise, the security and reliability needs for each data channel, and the pattern of aggregation and fan-out of high-volume business processes.

An important aspect of this step is to identify design alternatives that are under consideration, so you can explicitly measure the costs and benefits of each alternative based on the benchmark tests. For example, one design approach might involve receiving, validating and routing business requests in some proprietary text format, using existing relational DBMS tables to aggregate summary information. An alternative approach might be to convert the incoming text to a canonical XML format, which would allow validation and transformation using standard XML tools and a persistent XML service to store and aggregate the collection. With a well-planned benchmark test, you are able to weigh the functional pro's and con's of the two approaches in light of the measured costs and efficiencies.



Since the purpose of the test is to analyze the messaging/routing processes, you can generally collapse a large number of application scenarios into a much smaller set of messaging use cases. These use cases will define the range of message loads, routing patterns, service invocations and general quality of service needs.

3.3 CHARACTERIZING THE MESSAGING BACKBONE

Based on your planning analysis, you will know whether raw messaging performance will be a factor and will, therefore, be a controlled and tested aspect of the benchmark. If you think the messaging layer is important to your analysis, you must plan to set up a messaging infrastructure that will emulate the proposed production environment. This especially includes the use of broker clusters and specialized fault tolerance approaches, such as Sonic CAA. Once you have an idea of how many messages will be passed through the backbone, you may be able to perform a test of messaging brokers in isolation from the rest of the ESB infrastructure by simulating the raw JMS load expected. These results may allow you to perform smaller, more manageable tests of ESB infrastructure, on the assumption that you can scale these across different systems in the long run.

3.4 SIMULATING EXTERNAL FACTORS

The behavior of the projected ESB deployment is likely to depend on certain factors that are difficult to re-create within a performance test, such as user interaction, external data feeds, legacy applications or partner collaborations. Always keep in mind that the objective is to predict the ability of the ESB to scale to production levels without encountering unworkable bottlenecks. The simplest way to simulate an outside actor in your system without actually constructing it is to figure out the nature of the messages it produces or consumes and specify an endpoint within the test that behaves in that way. This could include large database applications, J2EE app servers or Web servers. Instead of hosting the actual subsystems, you can simply determine the nature of the message load going in or out. Then the Sonic Test Harness, described below, can help you simulate the message load these components are expected to generate.

You can also code up a more sophisticated simulation of an external service. This may be particularly important if that simulation will also add value to general quality assurance (QA) tests. This may also be necessary if there are patterns in message handling and content that are beyond the capabilities of the test harness. One technique used by Sonic solution engineers is to create a **stub implementation** of the process in which each service step for the test case is configured with simple pass-through service prototypes. This can help you get a baseline performance estimate for the ESB that is not



complicated by the full service implementations. It also makes it easier to diagnose runtime problems by eliminating the complexity of the full implementation. Sonic provides options for prototyping services, both in the regular product suite and in the library of custom services maintained by the Sonic consulting organization.

3.5 DEFINING INPUT LOAD

The load on the ESB is determined by the arrival of input messages at the various entry endpoints of the simulated business processes. An ESB provides optimal scalability and agility when input endpoints are loosely coupled with the services they invoke. Specifically, you should, to the extent possible, specify endpoint behaviors that:

- > Permit publication of input messages in multiple, simultaneous sessions
- > Allow for asynchronous delivery, acknowledgement and response
- > Constrain transaction scope to a single service invocation, limiting use of XA protocol

The specification of the input message load primarily depends on the rate of message production and the size of messages. Message arrival rate can be specified explicitly, or the test can simply attempt to increase the message load until the maximum limit is determined. The specific message rate generally requires configuring sleep time in the test harness or a prototype service and may also attempt to simulate variation and bursts in the message rate that are expected in production. For sophisticated integration scenarios, you may need to build a matrix with the projected message rate, size and configuration for each.

3.6 HARDWARE OPTIONS

The choices available for test hardware are often seriously short of the intended production environment, bringing us face to face with the "eschew extrapolation" rule. The answer is to partition the planned deployment environment into subsystems that can be independently tested with more modest test facilities. This allows you, for example, to take a routing scenario involving hundreds of servers and test the capabilities of each server type in isolation. The key, in such cases, is to deeply analyze the architecture of the solution, to validate to your satisfaction that there is no potential hidden bottleneck or shared resource that would prevent the production application from scaling linearly. This is a big assumption, so take the time to think it through.

An even simpler solution to the problem of limited test facilities is to set objectives for the benchmark test to produce results equal to the production requirements, but on the limited test hardware. It is almost never wrong to assume that things will run at least as fast when you host them on larger, faster systems; the risk is generally only in assuming they will scale up linearly.

Finally, some projects may have as a goal the investigation of specific hardware alternatives. These can include using solid state disks or network appliances for messaging logs, using an XML appliance instead of a software-based XSLT transformation service, or simply choosing a hardware architecture and operating system. If you are comparing hardware, you will have multiple test environments. The challenge will be how to interpret the actual cost/benefit of the options, given the performance results and the configuration options for each alternative.

3.7 MAKING THE SPECIFICATION REALISTIC

Of course, the only perfectly realistic test is to actually deploy the system into production, but that is also the worst possible moment to discover a problem in scalability or response time. Even a close approximation of the planned production system might be beyond reach. The key is to be realistic about those factors that have the highest potential to introduce variability and, therefore, risk. For example, while the production environment may involve thousands of client machines, each submitting messages at a slow rate, you can simulate this with a client emulator on one or more server machines, creating a comparable number of connections and generating the appropriate message load. As always, you need to write down and review the assumptions you make in justifying these simplifications.

If this is a competitive test among multiple vendors, you need to be especially prepared to make judgment calls concerning whether a particular manipulation of the solution or relaxation of requirements is realistic in light of your stated objectives. The best support, in this case, is for the original objectives to relate in a straightforward way to real business values, so that you can evaluate the impact of the modification on the ultimate goals. The time and effort spent attempting to keep low-level details in sync across a range of tests for several different vendors is better invested in rolling up your analysis of the **big picture** value of the solution and in cultivating an early, partnership relationship with each vendor.

Finally, consider non-performance goals for the benchmark. Common objectives in this area include availability, longevity, manageability and flexibility. If you don't address these during the performance test, you may discover down the line that the solution to problems in these areas may dramatically impact system performance, invalidating your original results.

How Vendors Cheat at Benchmarks...

- > Encourage tests within their own sweet spot
- > Provide default "eval mode" settings that improve speed by compromising integrity
- > Advise you to extrapolate results
- > Convince you it's "unfair" to accommodate other vendors' ideas about design

But Most Vendors Don't Cheat. They:

- > Provide feedback on your benchmark specification vis-à-vis real deployment needs
- > Explain the design tradeoffs inherent in their approach
- > Help tune their product for optimal results
- > Suggest alternatives in areas of weakness



4.0 BUILDING THE BENCHMARK

The performance project will be staged as a series of performance benchmarks. Each benchmark test is designed to answer specific questions in the spirit of scientific experimentation. As in science, the result of each test should confirm or disprove your hypotheses, narrowing the choice of optimal design alternatives, which then helps to define the next test. With a modest effort, you can outline the global objectives and questions to be answered, then sketch a brief plan for each test indicating what will be manipulated and what will be measured. Finally, you analyze and record the results, update the picture of what you know versus what you need to know, and zero in on objectives for the next round of benchmarks.

The modest time spent in preliminary analysis and in setting up test systems and programs is certain to be repaid in improved productivity. More important, the thought invested will help focus the tests on what really matters for your integration project. Given modern software standards, such as JMS, Web services and XML, it is also possible to build generic test programs that allow you to test many different design alternatives and topologies with minimal change of source code. Given sufficient automation of the test framework, you can combine the scientific test regimen with a rapid prototyping development approach and achieve stellar productivity in your efforts.

4.1 INDEPENDENT VARIABLES

The first test design question you should address is what factors you can manipulate for comparison. These include anything that you have control over, that is a realistic alternative in deployment, and that is likely to have bearing on the performance, scalability or reliability of the system. Some common candidates include:

- > The distribution of messaging servers, service containers and clients across machines
- > The mapping of client connections, sessions and threads to message destinations (topics and queues)
- > Message content and delivery options
- > Vendor/provider of messaging and/or service hosting technology
- > Message server clustering and failover options
- > Alternative implementations of key services, such as routing and transformation

4.2 DEPENDENT VARIABLES (MEASURES)

For each test case, the test software needs to measure key factors in the outcome, such as:

- > Throughput—the number of messages, service invocations or round-trip process executions that can be performed in a period of time
- > Latency—the time required for a message-delivery or complete service **round trip**
- > Exceptions—the count of failures, rejected messages or transaction deadlocks that occur during the test

These measures should apply directly to the cost/benefit objectives defined in the benchmark analysis. They enable you to scale the actual value of measured improvements in throughput, latency, simplicity or quality and, therefore, prioritize the design alternatives.

In addition, you want to measure **co-variants** of the results, specifically the finite resources being consumed from the underlying infrastructure. These measures are especially important if you want to do long-term resource planning for the integration infrastructure. Important co-variant measures include:

- > CPU usage and multi-CPU scalability
- > Memory working set
- > Thread contention and thread context switches
- > Network load
- > Disk I/O rates

4.3 CONTROL VARIABLES (CONSTANTS)

There are other variables that are not relevant to the benchmark objectives and that are best kept constant. These include factors you may have no control over, perhaps hardware network speed or the rate of an incoming feed. They may include choices that are already made and not, therefore, open to change, such as operating system, application server setup or message schema and size. Finally, they may also include



potential **artifacts** in the test that are not likely to occur in the production environment and are, therefore, undesirable in the performance simulation. You need to guard the quality of the information you gather by carefully controlling or avoiding artificial factors that can bias the test. Figure 4 shows some common ones we've seen in the Sonic Performance Lab.


Figure 4. Common Benchmark Artifacts

Test in a dirty state	For example, you may have queues blocked up from the previous run or uncommitted transactions that must be rolled back at the start of the test.
Program bug	When you get unbelievably fast response time for an operation, it might be that your code is skipping the operation altogether.
Misconfiguration of the system	One recurrent problem we see is that a recovery log file is located on an NFS mount, thus slowing performance and also compromising recoverability.
Underconfiguration	In most cases performance measures from a small system cannot be accurately extrapolated to server-class systems, and distributed performance can never be predicted from single-host performance.
Premature termination	If your test runs are too short, you may see transient performance related to initialization and in-memory cache that is not actually sustained in a longer test run.
Single-threaded client	Application code or poorly understood APIs from vendor software can inadvertently force each thread to serialize against the others, effectively destroying the chance for concurrent processing.

4.4 CONFIGURING SERVICES AND ENDPOINTS

To understand the performance characteristics of the complete system, the test implementation of channels and services must demonstrate realistic latency, system usage and routing patterns. For endpoints, this means that their protocol and security configuration will be similar in kind and complexity. For services, the main goal is to ensure that any significant resource consumption (e.g., memory or CPU) is allowed for, and that the fan-out and fan-in behavior of the production system is emulated.

For distributed performance tests, you may set up large message broker clusters with dynamic load balancing. This enables a true test of scalability, but will complicate the logistics of running tests. We advise spending extra time to map out the endpoint URLs, message connection factories and cluster configurations. In particular, if the vendor provides broker templates and container collections to assist in managing the various deployed processes, we highly recommend you take advantage of them in running your



benchmarks. Of course, based on the **build bottom-up** rule above, you'll want to do a preliminary test with a single broker prior to scaling up, so you can better quantify the scalability achieved.

4.5 DRIVING THE TEST WITH THE SONIC TEST HARNESS

You will also have to set up some mechanism to simulate the message sources and destinations for the services/processes being tested, i.e., the message producers and consumers that interact with the system being tested. For the test to be meaningful, you need some way of producing and consuming these messages with the configuration and load rates defined in the benchmark specification. It is important to do this carefully, since both message load and data content are largely determined by these components. To solve this problem, the Sonic Performance Lab has developed a flexible test harness, which we invite you to use directly or customize for your needs. The Progress® Sonic Test Harness is available online at <http://www.progress.com/sonic>.

It is composed of the following primary Java modules:

TestHarness—provides the program main; invokes Driver to run test

Driver—manages connections and launches ProducerObj or ConsumerObj threads to perform messaging operations

ProducerObj—implements a Producer thread; invokes a message generator and handles the warm-up cycle

ConsumerObj—implements a Consumer listener thread; monitors message size and latency

IntervalTimer—orchestrates test measurement intervals between main and messaging threads; aggregates counter data from all test threads

TestParams—manages configurable properties for the test; supports setting properties via input file, command line argument or user prompt

MessageGenerators—a suite of classes implementing the TestHarness.MsgGenerator interface, used by the ProducerObj to generate messages; allows you to control the type, size and content of messages

Utility programs—include the DiskPerf utility, which measures baseline disk performance, NetPerf, which tests network performance and NetPing, which is used by the TestHarness to correct for inconsistencies among system clocks



In brief, here is how the Sonic Test Harness code works:

- > Input parameters are parsed and evaluated, using the TestParams object.
- > A unique JMSCConnectionFactory object is instantiated using JNDI or (if that isn't available) a vendor-specific factory class.
- > A pool of JMS Connections is created and distributed among a pool of client threads, which are either ProducerObj threads or ConsumerObj JMS Message Listeners.
- > The threads and connections are started, and an IntervalTimer is used to signal the start of the test.
- > Each test thread performs message sends or receives, as configured, and stores timing data in a Collector object owned by the IntervalTimer.
- > At the end of each test iteration, the main program extracts timing information and outputs interval results.
- > After the last interval, the test end is signaled, and each thread cleans up; the TestHarness prints summary data and closes all connections.

The test harness comes complete with full documentation, which provides javadoc descriptions of each module and includes package summaries that describe the design of the test harness and suggested usage. There is also a set of example scripts and message generation templates.

Sample Sonic Test Harness Usage

<code>java TestHarness -help</code>	Display command options
<code>java TestHarness -props samples/props/simple_pub.props</code>	Use test properties from file
<code>java TestHarness -prompt</code>	Prompt user for parameter values
<code>java TestHarness mode=pub numsessions=4 sleep=2000</code>	Override several default values
<code>java TestHarness -props current.props -prompt</code>	Start with properties, but prompt for override values

4.6 CONFIGURING THE TEST

The Sonic Test Harness provides runtime parameters to configure the key messaging factors that are to be manipulated in the test run. This implementation conforms to the JMS 1.1 specification and will work with any JMS solution supporting that standard (using the JNDI interface to supply a vendor-specific connection factory). Each invocation of the test harness runs in its own JVM, with all threads acting as Topic Publishers, Topic Subscribers, Queue Senders or Queue Receivers. Thus, it may be necessary to launch several instances of the test harness to cover the entry and exit endpoints of a given test. It is also possible to configure the test clients to emit or consume reply messages.

The Sonic Test Harness allows a number of configurable parameters that can be modified to implement the independent and dependent variables of the test specification. In addition, it is very easy to add additional parameters when you customize the test harness. Here is a list of test harness parameters:

Basic JMS configuration parameters include:

mode	The JMS domain and role for this test; must be PUBLisher, SUBscriber, SENDer or RECEiver
numconnections	Number of JMS Connections used by this client
numsessions	Number of JMS Sessions in this client, same as number of threads
destname	Name used to generate topic/queue names
numdests	Number of topics or queues accessed by this client
destspersession	Number of destinations serviced by a session (must be <= numdests)
istransacted	If true, use transactions composed of msgpertxn messages
msgpertxn	Number of messages sent or consumed within a transaction; only applies if istransacted is true



Producer (i.e., publisher or sender) options include:

deliverymode	Delivery mode for message sender; standard JMS modes and Sonic extensions are all supported.
msgpriority	The JMS Message Priority to be assigned to the message
isreplyto	If true, the sender thread will wait for a reply message after sending.
replytodest	The reply topic or queue, or "tempdest" if a new temporary destination is to be used for replies

Message generation options are:

msgtype	The JMS Message Type, e.g., Text, Bytes, XML, etc.
msgsize	Size of message to be sent, e.g., "10K"
msggenclass	Java class implementing MsgGenerator used to generate messages for producers
msgproperties	Specification for generating JMS message properties (based on selected msggenclass)
msgtemplate	Template for generation of message data, if msggenclass is "TemplateMsgGenerator"
msgfiledir	Filesystem path (including wildcards) for input message data, if msggenclass is "FileMsgGenerator"
startmsgum	The starting value used by message generator to create unique key values
objectmsgclass	The serializable object used for the Object Message, if msggenclass is "ObjectMsgGenerator"
numobjects	The number of objects to include in each message, if msggenclass is "ObjectMsgGenerator"
msgcache	The number of generated objects to cache and repeatedly use, or "0" to regenerate every time

Consumer options include:

 durable 	If true, all subscribers are durable.
 ackmode 	Client acknowledgement mode, can be AUTO_ACK, CLIENT_ACK, DUPS_OK or TRANSACTED
 isforward 	If true, consumer will respond to the incoming message by forwarding a response message back.
 forwarddest 	The topic or queue to which reply is sent, or "UseReplyTo" to send to the JMSReplyTo destination
 msgselector 	Message selector for consumer, or "none" if no message selector is desired

Test timing and measurement options:

 sleepsecs 	Number of milliseconds client will sleep between JMS calls
 numrptintervals 	Number of test intervals executed before exiting
 rptintervalsecs 	Interval between reporting output in seconds
 dowarmup 	If true, start sending messages immediately; otherwise wait until timing intervals begin.
 nummessages 	Approximate limit on total number of messages to send before stopping.
 checklatency 	Compute message latency, for Consumer only
 checkmsgsize 	Report on message body size
 resultsfile 	File path for comma-delimited result output, or "none" if no output log is desired
 variablelist 	The independent variable to be included in the results file output
 clockhost 	The host machine running the NetPing servers, used as the time standard for latency measurement



Connection and user specs options:

jndiurl	The JNDI Provider URL used to instantiate the ConnectionFactory; if "none"; JNDI lookup is skipped.
brokerurl	Sonic-specific connection URL, used if jndiurl not specified, or if JNDI lookup fails
initialcontext	The JNDI Initial Context Factory class
factoryname	JNDI lookup name for the connection factory
domainname	Sonic Management Domain for JNDI lookup of a Sonic connection factory
user	User name for JNDI and broker connections
password	Password for connection user
clientid	Unique root used to generate identifiers for JMS connection client ID and durable subscriber user names


Additional properties can be defined by the user and passed through to custom message generators or the output results file. For more detail, consult the **User Guide** and API documentation that are provided with the Sonic Test Harness.

4.7 CUSTOMIZING THE TEST HARNESS

The most common customization of the test harness is to add or enhance a message generator. This can be essential if service logic requires specific content and that content cannot be sufficiently simulated using the provided generators. Customization is accomplished by simply writing a Java class that implements the `MsgGenerator` interface and specifying that class in the `msggenclass` parameter.

There may also be other independent variables that you want to manipulate within test harness clients. In most cases, you can add a new parameter to represent the variable within the `TestParams` class, then use that value to perform the desired manipulation.

If you are testing integration solutions other than Sonic ESB, you can use the test harness without change, assuming those implementations support JNDI-based specification of message factories. On the other hand, it may be easier to modify the `Driver.getConnection` method to return the vendor-specific JMS `ConnectionFactory` object. If the test messaging system does not conform to the JMS specification, you could replace the `ConsumerObj` or `ProducerObj` classes with similar ones of your own. In this case, you will also have to rewrite the `Driver.getConnection` method to manage connection objects and change the types of some objects.



A final potential customization of the test harness is to embed it in some other runtime environment. For example, you can wrap a test harness invocation as a JMeter test module, a Grinder module or even an ESB service. To see how to do this, review the code in `TestHarness.main`, and wrap similar logic inside the parent test module.

4.8 CUSTOM SERVICE SIMULATIONS

The simplest custom service you can implement is one that performs the loopback pattern: i.e., its only function is to pass the input message into the service outbox. This allows you to include actual process implementations that involve the service, but probably won't accurately simulate the resource or behavior of the real service. You can enhance the basic loopback service to perform general CPU or disk activity, so you can get a more accurate simulation of the system resources you expect the production service to consume.

When defining services that simulate expected production systems, we recommend that you avoid the trap of "doing it right" in the sense that it is probably not feasible to write production quality code. While you may recover fragments of code and algorithms from the prototype after you're finished, attempts to think ahead to production-class libraries are often frustrated by the rapid code/design changes of the performance test process. You are highly likely to end up discarding the test code and starting fresh anyhow, so you should focus your efforts on ensuring that the prototype is a good simulation rather than a production-ready subset. Also, it is common to switch deployment schemes in mid-stream, so the more streamlined your code library is, the easier it will be to deploy on the fly.

5.0 EXECUTING THE BENCHMARK

5.1 THE PERFORMANCE TEST CYCLE

Figure 5 illustrates the iterative cycle that is recommended for performance testing. This approach allows for the fact that your priorities will change as you better understand the risks and possibilities of various design and implementation options. It also allows you to fine-tune the resources and adjust the time spent on each phase, in case you start to run out of time on the project.

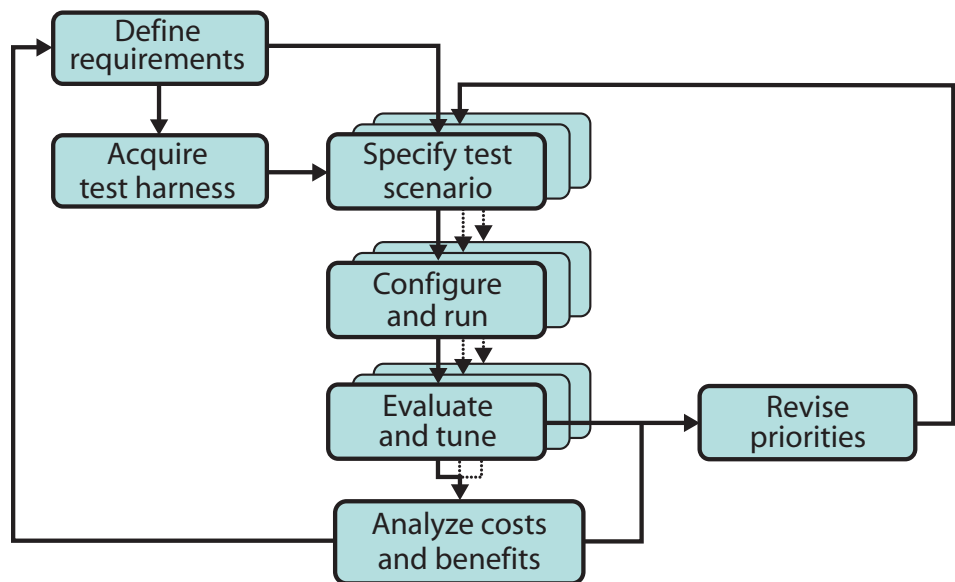


Figure 5. The Benchmark Test Cycle

Given that time and resources are limited, it is important to allow tasks and tests to run in parallel. You can make use of time spent waiting for long test runs to get responses from vendors, get decisions from team members to analyze earlier runs, or plan subsequent runs. You should recognize that the original test plan will change many times and leverage that fact to gain knowledge and improve the outcomes as you proceed. It is possible you will encounter bugs in the test programs or underlying infrastructure that you will have to fix or work around before proceeding.

5.2 CONFIGURING AND RUNNING TESTS

Once you have services and endpoints configured and ready to test, there may be a matrix of individual tests you need to try within a round of testing. For example, you might

set up one alternative implementation of a service and set up test harness properties files for the initiating and consuming clients, then want to try the test with varying message size, different ESB routing rules and differing numbers of sessions and listeners. In this case, it is advised that you save the test harness properties to a file and reuse these properties, overriding the specific values on the command line. For example:

```
java TestHarness -props cfg/RoutingTest.props mode=pub
numsessions=10 msgsize=10k msgproperties={route=A,user=X}
```

Normally, you start all message consumers prior to the message producers, to avoid an artificial backup in message delivery. If you have multiple test harness clients to run, you can open command windows for each, queue them up to run, then hit the <enter> key in each window to start them. For large numbers of test harness instances it would be desirable to embed the test harness in some distributed test management environment (see *Customizing the Test Harness* above). As the test runs, you will want to verify that the messages leaving the system correspond in number to the messages going in and also monitor service container logs to verify no exceptions are encountered. You will also want to monitor and record levels of memory and CPU usage. The test harness does not validate results, so a certain amount of diligence is called for to verify that the test is performing the functions it is supposed to.

The test harness is designed to be *idempotent*² with regard to the messaging infrastructure; that is, it cleans up queues, removes durable subscribers and otherwise tries to leave the system in the same state as it was initially. You need to perform a similar analysis on any services you repeatedly invoke across tests. One good example of this is if you insert rows into a database that include a unique key, you either need to modify the key values used on successive runs (the test harness parameter *startmsgnum*, combined with *msggenclass=TemplateMsgGenerator* can accomplish this), or you need to delete the unique rows after each test. It is also possible that services, brokers or other servers could change state during a test in a way that changes the results for subsequent tests. Ideally, you bring all such services into a *warm* state and then assume they will remain stable across a sequence of tests, but in the worst case you may want to restart the services. In very rare cases you may want to reboot the system, run a brief *warm-up* test, then validate that performance is predictable.

² En.wiktionary.org: idempotent-describing an action which, when performed multiple times, has no further effect on its subject after the first time it is performed.

5.3 COLLECTING AND EVALUATING RESULTS

If you are using the Sonic Test Harness to drive the tests, each test client will print out the values of all test parameters, then give you a summary of throughput and latency. Here's an example of that output:

```
[07/03/24 19:27:28]: Test started.
```

```
-----  
Interval #  
|  
|           Interval Message Rate (msg\sec)  
|           |  
|           |           Interval Msg Count  
|           |           |  
|           |           |           Cumulative Msg Total  
|           |           |           |  
-----  
Interval 1      1265      12658      12658  
Interval 2      1377      13781      26439  
Interval 3      1353      13537      39976  
Interval 4      1381      13804      53780  
Interval 5      1347      13489      67269  
Interval 6      1304      13047      80316  
Interval 7      1387      13907      94223  
Interval 8      1356      13575      107798  
Interval 9      1351      13523      121321  
Interval 10     1352      13527      134848
```

```
[07/03/24 19:29:08] Finished.
```

```
---Testing intervals finished.---
```

```
Ignored 3744 messages during shutdown
```

```
Overall Message Rate = 1347 msg/sec
```

```
Total Messages During timing intervals = 134848.
```

```
Total messages across run = 138592.
```

```
Average Message Latency = 20 (count=137700, min=0, max=1302)
```

```
Average Message Size = 6293 (count=137700, min=2883, max=8775)
```

Based on the output, you may see that Interval 1 is substantially slower because of first-time effects in the software. You could avoid this by setting the TestHarness parameter **dowarmup=true**, which would instruct test threads to perform initial warm-up iterations prior to the start of test iterations. If you do not see a leveling in message rate by the end of the test, you should consider increasing test length until you achieve **steady-state** performance. If you are going to perform a connected series of tests with multiple instances of test harness clients, you should consider appending test result data to an output file by specifying the **resultsfile** parameter with the name of the shared file used to collate results. A common technique is to add a user testID parameter to the list specified in the **variablelist** parameter. For example, if you set:

```
"variablelist=testID,mode,numsessions,deliverymode"
and "testID=test1", etc,
```

your results file would contain comma separated values that could be imported into a spreadsheet to look something like Figure 6.

Figure 6. Sample Result File from Test Harness

testID	mode	numsessions	numdests	deliverymode	TotalMsg	MsgRate	AvgLatency	MinLatency	MaxLatency
test1	pub	1	1	PERS	97098	970			
test1	sub	1	1	PERS	96703	966	0	0	163
test2	pub	10	10	PERS	183103	1830			
test2	sub	10	10	PERS	1825169	18231	16	5	355

A second source of data should be system monitors on the various test hosts, tracking the CPU, memory, context switching and disk usage on the machine. For Windows, either the task manager or the perfmon utility can provide an adequate snapshot of resource usage. For UNIX hosts, there is a choice of similar graphical tools, or you could simply use the built-in top command; one syntax that we find works well is:

```
top -Cbi -d 10 -n 20 | tee top.out
```

Note that the -d (delay in seconds) and -n (number of iterations) params should be consistent with the time intervals you specify for the test harness, to make it easier to correlate data.




5.4 TUNING AND TWEAKING

As you execute the benchmark, you can investigate several tuning options, including:

- > Client and service connections, sessions, producer threads and message listeners
- > QoS settings (especially Send mode, Ack mode, transactions)
- > Log file disk and use of non-persistent replicated mode
- > ESB intra-container messaging option
- > Message broker buffer sizes (there may be several of these)
- > Java heap size (i.e., the `-ms` and `-mx` parameters of the Java command)
- > ESB transformation options, including *xcb*r routing configurations
- > Broker security settings

Tuning client and service code is more open-ended, as it can easily involve tradeoffs between functionality, reliability and scalability. There is no way to list all possible ESB service tuning techniques, but here are a few we've found useful:

- > Move important data fields from the message body into message properties, so you can do routing and database lookups without parsing XML.
- > Bundle related services in the same container, and turn on intra-container messaging to avoid messaging overhead.
- > Manage SOAP messages from a reliable source as HTTP Bytes messages to avoid validation and envelope overhead.
- > For slowly changing, highly used database information, cache results in memory and refresh at regular intervals.
- > Use a text parsing service, rather than an XML Transform service, to split batch messages into records.

-
- 
- > Use the Sonic Large Message Service instead of an FTP service for very large messages.
 - > Associate external endpoints with a secure broker in the DMZ and internal endpoints with non-secure brokers behind the firewall.
 - > Use the built-in JMS pub/sub mechanism for large message fan-outs, rather than an ESB process.
 - > Use JMS clients instead of SOAP clients wherever possible, and always cache and reuse connections, sessions and temporary queues.

5.5 ANALYZING THE COST/BENEFIT OF TRADEOFFS

It would be nice to collect all data before analyzing the tradeoffs and feasibility of design options, but the realities of time constraints may force you to select the likeliest alternatives early in the process, so you can invest more time investigating them in depth. Thus, the recommended approach is to take stock after each test series and modify test priorities based on outcomes. For example, if performance becomes completely unacceptable at a certain load level, there's no point in testing even higher load levels. On the other hand, if speed or throughput approaches the theoretical best-case, there may be no point in attempting further tuning.

The test approach outlined in Figure 5 encourages constant reevaluation of test scenarios and variables, based on new insights gained as testing proceeds. If you have the luxury of a sophisticated ROI analysis for the project, you may even be able to project the real value of scalability (measured in messages per second, connections per host or latency of response time) in terms of actual business value, if not monetary value. This makes it easier to make quick decisions on which alternative should be selected, based on test results, which, in turn, allows you to explore the preferred alternative in more depth. The challenge here is to invest just enough time to make certain your subsequent test runs will be productive without getting lost in analysis.

Here are some examples of tradeoffs you might be called on to analyze during an ESB benchmark project:

- > Increasing the number of CPUs on a host versus adding more hosts
- > Better response time versus higher total throughput

What is "Fair"?

Sonic performance specialists have witnessed scores of benchmark projects comparing multiple integration vendors where there was a strong determination to be fair and equal to each of the participating vendors. While the natural human tendency toward fairness is understandable and commendable, it is our considered opinion that it can lead to practices that are undesirable from a business point of view and perhaps even to unintended bias in the tests.

Why a "fair" test may not be the right thing:

- > Unless you are a public organization with specific procurement guidelines, there is seldom any legal or ethical requirement for a business to treat all potential vendors the same. In fact, vendors have different strengths, and you should focus on those factors that are most relevant to how the specific vendor could work with your company in the future.
- > Efforts to ensure absolutely equal treatment often lead to

- > Smaller memory footprint versus faster cached XML access
- > Higher message throughput versus faster persistent message recovery time versus additional hardware
- > Better client performance and fault tolerance versus cost of deploying vendor specific client code
- > Simpler infrastructure and configuration versus performance
- > Increased message security versus faster response time

6.0 PERFORMANCE ENGINEERING AND COST/BENEFIT

Most vendors know that they can guarantee they can win a performance benchmark against all competitors if they can write the specifications for the benchmark. The reason is that you can almost always make something run faster if it does less, and each vendor has their own set of design tradeoffs and unique "sweet spot." But for the customer who is comparing vendors there are limits in what can be sacrificed to achieve performance, especially in the areas of robustness, productivity and security. Performance and scalability can be key indicators of the feasibility of a solution, but there are many other costs and benefits that must also be balanced.

In particular, when deploying an enterprise messaging solution, you should be sure you analyze the relative cost and benefit of:

- > Performance improvement achieved by purchasing additional hardware
- > Solutions for ensuring fault tolerance
- > Training required for development and operations staff
- > Manpower and time required to build and deploy the solution
- > Effort and time required to implement changes, once the system is deployed

6.1 PROGRESS' COMMITMENT TO SCALABLE SOLUTIONS

At Progress Software, we recognize the complexities of conducting benchmarks, and apply a broad base of rigorous tests to our software throughout our development cycle to ensure that Sonic ESB not only performs well in specific tests, but also excels in many of the real-world scenarios for which it will be implemented by customers.

From the start, Progress' development team has placed an equal value and effort on performance and scalability as it has on features; after all, performance and scalability are features and extremely important on their own. Sonic has always had a dedicated performance team that is responsible for constantly measuring and improving performance and scalability, release after release.

As part of this commitment to performance and scalability we have created an extensive benchmarking infrastructure in our testing lab. To this end, our commitment is backed up with a methodology, an experienced performance engineering team, dedicated lab hardware and extensive software test technology. This test environment enables us to deliver a product that is time-proven to be able to cope with the demands of the real world integration environment. We apply performance tests based on the Sonic Test Harness and our more advanced load test framework in our testing lab across the array of hardware described above. Our standard product test suite includes testing of the product with varying numbers of senders, receivers, queues, topics and servers and with realistic message loads. Test results are analyzed to correct any regression in performance from release to release and to identify new areas for work.

In addition to performance numbers, the Sonic ESB product test suite exercises and validates real-world reliability. For example, we test the order in which messages were received, the number of times a particular message gets received under various queue and pub/sub scenarios and any loss of messages as they are being delivered to their intended destination. (The JMS specification allows message loss under certain circumstances, but in the real-world, is best avoided.) Without such measurement and detection, it would be impossible to know how a distributed messaging environment is really going to behave under a heavy load. Regardless of how well a middleware product seems to be architected, you can never be really sure about these real-world scalability and reliability issues unless you test it under stress.

rigid benchmark specifications with little leeway for testers to diverge; this negates the most profitable aspect of benchmark testing, which is knowledge gained from trial and error. With a rigid spec, you miss the considerable insight into architectural issues, design tradeoffs and real world factors that you would gain with a more dynamic approach.

- > All vendors are not equal, and performing the complete matrix of tests, even after it is clear who the leaders are, requires considerable additional effort. Of course, every vendor can make a reasonable case for its own solution, so your goal should be to partner with each one to discover the best and most appropriate use of their technology. Although this may be a different path with each vendor, you will be comparing the best fit solution in each case, and you will also gain valuable insight into how the vendor works with customers in general.

Bottom line: you own the test and it's your own best interests that matter.



7.0 CONCLUSIONS

Benchmarking an integration solution is a complex task that requires careful planning and execution if results are to be meaningful. It is important to go through a proven methodology of analyzing objectives, specifying test requirements and iteratively testing and evaluating hypotheses, very similar to the classic scientific method of investigation. By conducting such a benchmark you can not only be confident of meeting your original goals, but also of learning a great deal about your application needs, production environment and integration tools.

Use of an ESB may reduce the time needed to develop and stage tests, but it also increases the number of design and architectural options you may wish to evaluate. If you find at the end of the test project that you have answered all your original questions, but that the list of new questions is longer than the original one, don't be disappointed. That's simply a sign of the richness and flexibility of the ESB approach.

A methodology alone cannot guarantee success. To ensure your time is well spent, you also need to involve people with the necessary skills and experience. This particularly includes experts in the application requirements, in performance tuning and testing, and in the integration infrastructure. For infrastructure products, such as an ESB or messaging subsystem, you should get the vendors involved and give them an opportunity to teach you the best way to use their product. We are confident that your investment in people and time will more than pay off in savings down the line, based on the informed decisions you can make about when and how to proceed.

ABOUT THE AUTHOR

David Hentchel works as a Principal Solution Engineer for Progress Software. He has spent much of his 30-year career in performance skills training, benchmark testing and application tuning. Initially working on mainframe structured database and OLTP solutions, he had early involvement with object databases, XML databases, Java Messaging Service and ESB technologies.

PROGRESS
SOFTWARE

Worldwide Headquarters

Progress Software Corporation,
14 Oak Park, Bedford, MA 01730 USA
Tel: +1 781 280-4000 Fax: +1 781 280-4095
www.progress.com

For regional international office locations and contact information, please refer to www.progress.com/worldwide

©2007 Progress Software Corporation. All rights reserved. Progress is a registered trademark of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and other countries. Sonic, Sonic BPEL Server, and Sonic ESB are trademarks or registered trademarks of Sonic Software Corporation or one of its affiliates or subsidiaries in the U. S. and other countries. Any other trademarks contained herein are the property of their respective owners.

prod. code 3991



0000113911