WRITE ONCE.
SCALE ANYWHERE.

The Scalability Revolution: From Dead End to Open Road

An SBA Concept Paper

By Nati Shalom, CTO | February 2007

Abstract

Application workload is growing at an increasing pace, making scalability a prime concern of application designers and administrators. In this paper, we define scalability, and show that inherent scalability barriers represent a dead end for today's tier-based business-critical applications. We argue that in order to survive, these applications must achieve *linear scalability*, and that the only way to do this is to switch from the tier-based model to a new architectural approach. We suggest a novel approach in which applications are partitioned into self-sufficient processing units, and present Space-Based Architecture (SBA) as a practical implementation of this approach. We demonstrate that SBA guarantees both linear scalability and simplicity for designers, developers and administrators—transforming scalability from dead end to open road.

Table of Contents

Executive Summary

In many application domains today, especially in financial services, the number of clients, the depth of services provided, and the data volumes are all growing simultaneously; in parallel, middle-office analytics applications are moving towards near-real-time processing. As a result, **application workload is growing exponentially**. One GigaSpaces customer is expecting to grow from 100K trades to 80 million trades—in only two years!

In order to understand the scalability problem, we must first **define *scalability*:** scalability is the ability to grow an application to meet growing demand, without changing the code, and without sacrificing the data affinity and service levels demanded by your users.

We identify two situations in which scalability is interrupted:

- A ***scalability crash barrier***—occurs if your application, as it is today, cannot scale up without reducing data affinity or increasing latency to unacceptable levels.
- A ***marginal cost barrier***—occurs when the cost of scaling your application progressively increases, until scaling further is not economically justifiable.

For most contemporary applications, particularly transactional applications in a low-latency environment, these barriers are inevitable. But this is not the only possible case. Theoretically, an application can achieve ***linear scalability***—the ability to grow as much as needed, at a fixed price per capacity unit—in which case it would never face scalability barriers.

Tier-Based Business-Critical Applications—Scalability Interrupted

Consider two typical business-critical applications—front-office applications and back/middle office analytics applications. It is clear that both types of applications **do not have linear scalability**, because scaling becomes progressively more difficult and expensive as the application grows.

Interestingly, these two very different business-critical applications have striking similarities: they are both **stateful**, and both use a messaging tier for coordination, a data tier for storage of state information, and a business tier for the actual processing—in other words, they are both founded on the ***tier-based architecture***.

More interesting still, both types of applications encounter **similar scalability problems**: cluster nodes get overloaded by inefficient clustering; different clustering models for each tier cause unnecessary ping-pong inside the tiers; unknown scaling ratios between system components cause unexpected bottlenecks when the system scales; growing messaging volumes might overload the processing components; the network becomes the botleneck; inability to virtualize the tiers causes coordination problems; and different H/A models for each tier makes it difficult to guarantee recovery from partial failure.

Each of these problems can cause a scalability crash barrier. To avoid hitting a crash barrier, application administrators are forced to apply temporary fixes—complex, resource-consuming coordination and clustering mechanisms—and scale up each tier just to accommodate the additional overhead. As the system scales, these fixes need to be applied again and again, making scalability progressively more expensive.

The application's administrators find themselves **caught between a rock and a hard place:** if they apply the scalability fixes, the application grows more and more complex, until it hits a marginal cost barrier; but if they don't apply these fixes, the application quickly hits a scalability crash barrier, and must be replaced.

This dilemma is inherent in tier-based applications: because the system is divided into separate tiers, it increases in complexity as it scales, requires more and more overhead just to manage this complexity, and makes it more and more costly to increase capacity.

Therefore, **tier-based applications *cannot* be linearly scalable**. This is proven by the well-known *Amdahl's Law*, which states that if all processors in a system spend some of their time on overhead—as is the case in all tier-based systems—the speed improvement yielded by additional processors quickly hits an upper boundary (for 10% overhead, maximal improvement is 10X).

If scalability is a road, and applications are cars driving on the road, **non-linear scalability is a dead end**. To avoid ending up as a wreck, tier-based business-critical applications must become linearly scalable—and the only way to do this is a change of architecture.

Towards a Linearly-Scalable Architecture

The key to a linearly-scalable architecture is to **expand the application by adding self-sufficient units**. This way, as the application grows, there is no complex coordination that consumes resources and leads to scalability barriers.

This is a clear departure from the tier-based architecture: instead of running each application tier in a separate cluster of computing resources, all the tiers are compressed into a single unit, which is duplicated when the application is scaled. **The middleware problem simply evaporates** when each component becomes self-sufficient.

But **in a stateful environment**, how can you manage a workflow when each machine is completely self-sufficient? How can machines share state information between them? The secret is to *collocate* all steps of the business process—putting them on the same machine, in the same JVM. This requires developing **a *processing unit***—a mini-application which can perform the entire business process on its own and return a result.

The processing unit manages its own workflow, providing messaging and data storage facilities to its collocated service instances. This saves the need to contact external resources, and means that **all process information can be stored in local memory**, reducing latency to a minimum.

To ensure reliability, state data and partial results can be persisted to a database, or replicated to another, identical processing unit. Thus, the processing unit **can be made as reliable as needed**: transient data can be stored in memory only; very sensitive data can be persisted to a remote database. This improves on the tier-based model, which forces all processing components to pay the high price of persistency—even if this type of reliability is not really required.

In the processing unit model, user requests that enter the system are distributed between the processing units using **content-based routing**—if the request has value A it goes to Processing Unit 1; if it has value B it goes to Processing Unit 2. This type of routing is very simple and inexpensive, but most importantly, it **does not increase in complexity** as the application scales.

The concept of a processing unit makes linear scalability possible, but this is not enough. A critical requirement is that as the distributed application scales, it should **behave and look as one server**, whether viewed by designers, developers, administrators, or clients. Everyone involved should see the system as one coherent unit: scaling should be simplified to the point of transparency.

Space Based Architecture—Scalability as Open Road

Space-Based Architecture (SBA) is a way to implement processing units in your application, transforming scalability from dead end to open road.

At the core of an SBA processing unit is the *space*—a middleware infrastructure, collocated with the business process services, which provides in-memory messaging and data storage facilities for the processing unit. Like a conveyor belt in a production line, the space helps coordinate the business process, and allows business services to deposit their partial results on a shared work area, which can be accessed by other services in turn.

The space's messaging facility allows **direct in-memory access to specific messages**—this saves each business process service the overhead of connecting to an external messaging server and screening out unneeded messages from the queue.

When the space acts as data store, the services can **save results at in-memory speeds**, because they are collocated with the space on the same physical machine.

The space has a built-in clustering model, which allows it to share data and messages with other spaces on the network; this allows processing units to guarantee reliability by **replicating state data between them**. The space can also **persist its data to a database**. All this occurs in the background, without affecting the latency of the processing units.

This unified clustering model allows clients of the application to **execute requests against a generic cluster proxy**—the cluster matches the request at the object level, and routes it to the appropriate space. This routing is done in a distributed manner, without requiring a central server.

A central part of SBA is **a deployment framework** that takes care of running the processing unit on physical machines. The framework consists of *SLA-driven containers* that run on all the computers in the deployment scope.

Via the user interface of the deployment framework, it is easy to **deploy as many processing units as needed—with one click**. A general *deployment descriptor* describes a generic scaling policy for all possible deployment sizes, so deployment doesn't become even a bit more complex when scaling up. What's more, the framework dynamically responds to application workload.

Because of the flexibility of the deployment framework, SBA permits **a combination of scaling-up** (within a multi-core machine) **and scaling-out**, without breaking the SOA model.

SBA **guarantees linear scalability, while ensuring *simplicity***—the application's scale becomes transparent to all involved, as if it was a single server:

- Scalability **ceases to be a design consideration**.
- The application presents a unified API and clustering model, so **developers can write code as if for a single server**—physical distribution is completely abstracted.
- The entire application is deployed through **a single deployment command**.
- Clients can perform operations on **the entire cluster in a single operation**.

The SBA Value Proposition

SBA removes the scalability dead-end facing today's tier-based applications, and guarantees improved scalability in four dimensions: (1) fixed, minimal **latency**; (2) low, predictable **hardware/software costs**; (3) reduced **development costs**; and (4) consistent **data affinity.**

1. The Scalability Problem

This chapter introduces a general problem facing business-critical applications today: the need to scale up and down continuously, quickly, and unpredictably. In today's business environment, it isn't enough to know how to scale your application, it is necessary to understand your application's entire scaling path—how far it can scale, how difficult it will be to scale in the future, and what barriers stand in the way of future scaling.

Section 1.1 sets the background of the scalability problem—application workload is growing **exponentially and unpredictably**.

Section 1.2 explains **what *scalability* really means**, and which barriers can stand in the way of scalability.

Section 1.3 defines ***linear scalability***—an application's ability to scale as far as needed at a fixed cost per capacity unit.

1.1. Faster Growth When You Least Expect It

Scalability was always an issue for business-critical software, but rarely a showstopper. Applications had a fixed and moderate rate of growth (e.g., 20% per year), so it was quite easy to provision resources in advance to meet the expected growth. For example, if the application was handling 100K trades today, you would provision enough machines to support 140K trades, and forget about scalability for the next two years.

But things are changing. In many application domains today, especially in financial services, the number of clients, the depth of services provided, and the data volumes are all growing simultaneously. In addition, batch-oriented analytics applications in the middle office are moving towards near-real-time analytics.

As a result, **application workload is growing exponentially**. One GigaSpaces customer is expecting to grow from 100K trades to 80 million trades—in only two years! This customer simply cannot provision resources in advance, because this means purchasing 800 times more resources than are needed to support the current workload.

And what if growth estimates are wrong? Today's multi-dimensional usage expansion makes it much harder to forecast future growth in application workload. This makes the problem even worse: not only is usage growing quickly, **it is also growing unpredictably**. Even if you had the money to purchase a huge pool of resources in advance, you might be underestimating actual growth, so you run the risk of downtime or unacceptable latency. Or you might be overestimating, which means you'll have wasted money on unnecessary machines.

In the wake of these changes, application designers and administrators must wake up to the problem of scalability. You need to ask yourself, "is my application scalable?" But first of all—what is scalability?

1.2. Defining Scalability and its Limits

Having ***scalability*** means that your application can increase its working capacity while keeping three things constant:

- **Consistency** of the business logic and data—otherwise when you scale, things won't work. (If scaling compromises reliability, but to an extent that your users can tolerate, it can still count as scaling.)
- **End-to-end latency**—otherwise when you scale, you might be supporting more clients, but degrading the service level each client receives. (If scaling degrades performance, but the service level is still acceptable to your users, it can still count as scaling.)
- **The application code**—otherwise you aren't scaling the application, you're replacing it.

So when you ask "is my application scalable?" you're asking whether your application, as it is today, can meet growing demand without breaking, and without sacrificing service levels, as demanded by your users.[1]

Scalability Crash Barriers

Our definition of scalability immediately implies several situations in which it is no longer possible to scale up:

- **Consistency barrier**—occurs when you cannot scale the application any more without threatening the level of business logic reliability demanded by your users.
- **Latency barrier**—occurs when you cannot scale the application any more without increasing *end-to-end latency* to an unacceptable level.
- **Coding barrier**—occurs when you cannot scale the application any more without significantly rewriting the code.

These three are ***crash barriers*** because when you hit them, there's nothing you can do to support the increasing workload, short of completely replacing your application.

This is illustrated in the diagram: The application can increase capacity by adding more hardware units, until it hits the crash barrier—and then the scalability curve flattens out, because no amount of additional hardware will allow scaling up, without compromising on required consistency or latency, or changing the code.

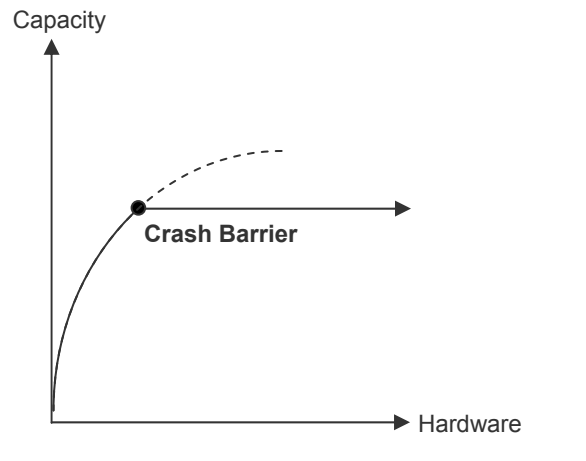


Figure 1 – Scalability Curve with Crash Barrier

[1] A similar definition is proposed by technologist Scott Burkett: "scalability refers to an system's ability to handle increasingly heavier loads from users' (activity) without [...] slowing down, or worse, breaking down completely."
(in *Scaling Your Technology with Your Business*, sourced from:
http://www.scottburkett.com/index.php/technology/2007-01-08/scaling-your-technology-with-your-business.html).

Scalability and End-to-End Latency

When we say that scalability is tied to latency, we are referring to the real latency that users experience—**latency for the entire business transaction**, which spans all application tiers.

Users don't care how long it takes for their request to pass through the messaging queue, or how long it takes for their data to be retrieved from the database. They care **how long it takes for the system as a whole to respond**. This is what we refer to as *end-to-end latency*.

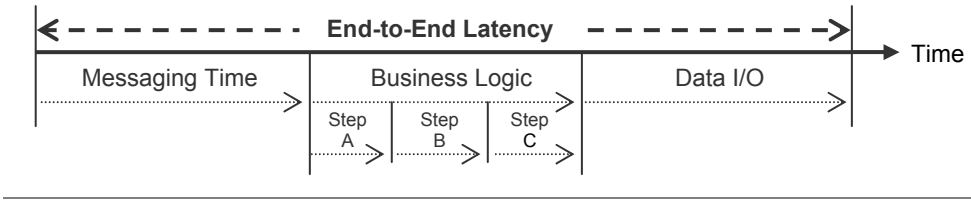


Figure 2 – End-to-End Latency—Spanning All Application Tiers

Very often, latency is measured separately for each tier: latency for messaging, latency for the business process, latency for data I/O. This doesn't reflect the actual latency that users experience, because it ignores the complex dependencies between the tiers, a complexity which only increases as the application scales.

Per-tier latency also hides the fact that **the system is as latent as its weakest link**. The slowest component—the bottleneck—determines the latency of the entire application. Latency improvements in other components may not improve end-to-end latency at all—they might even degrade overall performance, by increasing the load on the system's weaker links.

Technologist Scott Burkett agrees that in context of scalability, the relevant measure of performance is an end-to-end measure: "An application (or platform) is considered scalable if it can continue to service additional users […] without seeing any significant performance hit **from the user's standpoint**."[2] (emphasis added)

Difficulty of Scaling—the Marginal Cost Barrier

Now that we understand what it means to scale, we can ask a more important question: how difficult is it to scale? If you need to double your capacity tomorrow, while maintaining current latency and consistency levels, will you need to provide **twice the resources** you have today, **or ten times the resources**? And the next time you scale, will it become even more difficult?

The difficulty of scaling can be better understood by defining a first and second derivative of scalability:

- ***First derivative of scalability*—**how many units of hardware do you need to add to get one additional unit of capacity?
- ***Second derivative of scalability***—as you scale, does each additional unit of capacity require a bigger increase of hardware resources, and by how much?

[2] Scott Burkett, *Scaling Your Technology with Your Business*, sourced from http://www.scottburkett.com/index.php/technology/2007-01-08/scaling-your-technology-with-your-business.html.

Most real-life applications have a diminishing return to additional hardware—the first derivative of scalability is positive, and the second negative. The diagram shows two such applications.

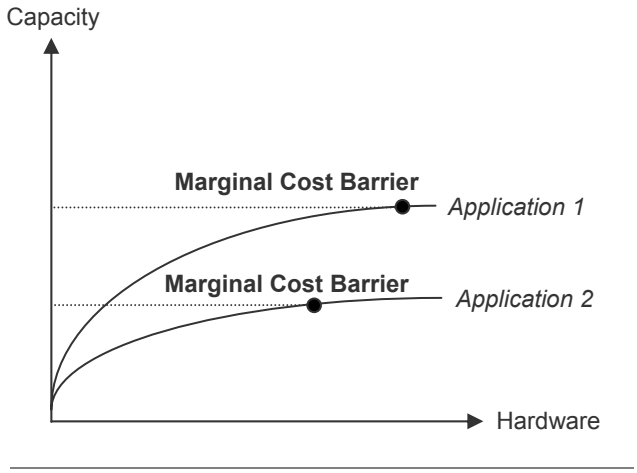


Figure 3 – Scalability Curve with Marginal Cost Barrier

These applications scale relatively cheaply at first, but **scaling becomes more and more expensive as they grow**. This characteristic, of diminishing returns to additional hardware, implies that these applications eventually reach a point at which is no longer economically justifiable to scale up. Application 2 reaches this point earlier, because at each point along the scaling path, it requires more resources to add the next capacity unit, compared to Application 1.

A situation in which scaling is possible, but the next capacity unit is too expensive to justify, can be called a ***marginal cost barrier***. Here it is possible to scale in principle—it's just not worth it.

Is the marginal cost barrier inevitable? For an application with diminishing returns to hardware, yes. But **this is not the only possible case**. Theoretically, an application might have constant returns to additional hardware, and in this case, it would never hit the barrier.

1.3. Linear Scalability—No Barriers

The following diagram shows an application which has constant return to additional hardware. The arrow at the end of the scalability curve shows that it continues indefinitely—the application can scale as far as required, at a fixed cost per capacity unit. In other words, the application is *linearly scalable*.

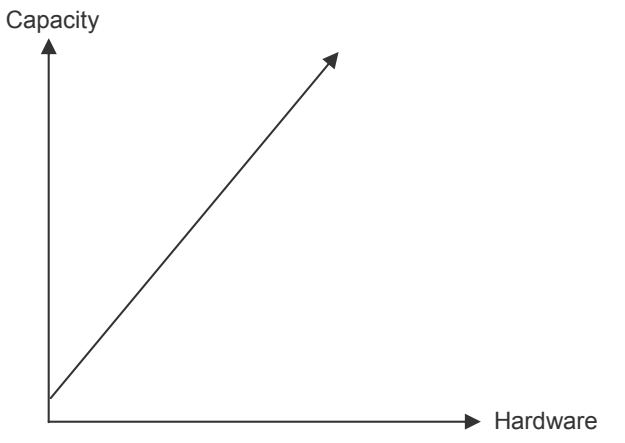


Figure 4 – A Linear Scalability Curve

This leads us to a basic definition:

> ***Linear scalability*** occurs when each new hardware unit always contributes the same amount of additional capacity.

Another way to define linear scalability is in relation to the scalability barriers we discussed earlier:

> ***Linear scalability*** ⇔ IF AND ONLY IF there are no scalability barriers at any point along the application's scaling path.[3]

The second definition is true because:

- If an application faces a marginal cost barrier, this must mean it has diminishing returns to hardware, which means it is not linearly scalable.
- If an application faces at least one scalability crash barrier, this means it has a 'broken' scalability curve (like in Figure 1), so it is not linearly scalable throughout the scaling path.
- If an application is linearly scalable, it **never faces a marginal cost barrier**—if you can afford the second capacity unit, you will also be able to afford the 200th unit.
- If an application is linearly scalable, it never faces scalability crash barriers—the application can **scale up indefinitely**, without compromising consistency, latency, or the existing code.

[3] This does not hold if the application faces a cost barrier along the entire scaling path—in this case it is not economically feasible to even start scaling up the application, so it doesn't matter if it is linearly scalable or not.

2. Tier-Based Business-Critical Applications—Scalability Interrupted

Now that we've defined scalability, we can discuss the scalability features of some real-life applications. Take two business-critical applications—**front-office** applications and **back/middle office analytics** applications. What does their scalability look like?

To start with, it is clear that both types of applications **do not have linear scalability** (as defined above). It is typically not possible to add more machines to the application and keep it working properly; and if scaling out is possible, it becomes progressively more difficult and expensive as the application grows.

Consider what non-linear scalability means. It means that sooner or later, **these applications must hit one of the scalability barriers:** either scaling will become so expensive that it will not longer be justifiable, or the application will simply break at a certain scale, and stop delivering the latency and consistency users need.

If scalability is a road, and applications are cars driving on the road, **non-linear scalability is a dead end.** And with the rapid growth in demand, drivers are pushing their accelerator pedals down to the floor, hurtling faster and faster towards the barrier.

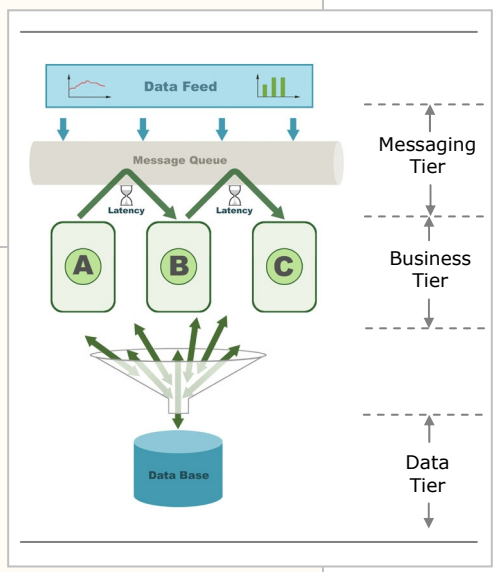
The conclusion is simple: To avoid ending up as a wreck, business-critical applications **must become linearly scalable**. In this chapter, we explain why the only way to do this is a change of architecture.

Section 2.1	shows that both front-office systems and back-office analytics—two very different types of business-critical applications—have **a similar tier-based architecture**.
Section 2.2	explains the **scalability barriers** common to both types of applications, and shows that these barriers stem directly from their architectural characteristics.
Section 2.3	proves that today's front-office and back-office analytics applications—as tier-based applications—**necessarily suffer from non-linear scalability**, and inevitably run into scalability barriers.
Section 2.4	concludes that **the tier-based architecture is the real bottleneck** of today's business-critical applications, and that only a change of architecture can transform scalability into an open road.

2.1. Front-Office, Back-Office—Similar Architecture

Front-office and back/middle-office analytics systems are very different in an applicative sense, and face different scalability challenges—but a closer look shows interesting similarities. Compare the typical business requirements and application workflow of both types of applications:

Type of Application	Business Requirement	Application Workflow
FRONT-OFFICE Typical examples: ▪ Market data, OMS/trading applications (financial sector) ▪ Pre-paid systems (telecom sector) ▪ Online reservations (travel sector)	**Process large volume of transactions** in near-real-time, with low latency, ensuring high business process consistency. The process is usually transactional—for each user request several operations must be performed in sequence, e.g. A, then B, then C.	1. User request arrives in an external feed or through client applications. 2. The request enters the messaging queue. 3. The messaging server queues the software component responsible for the first business process step, e.g. parsing (COMPONENT A). 4. COMPONENT A reads relevant data from the database (e.g. a text-based user request). 5. COMPONENT A does its work and saves the result to the database (e.g. the parsed request). 6. The messaging server queues COMPONENT B, and so on… Until business process ends. 7. The messaging server delivers the result to the clients (or queues them to read it from the database).
BACK/MIDDLE-OFFICE ANALYTICS Typical examples: ▪ Profit and loss calculation, reconciliation, value-at-risk (financial sector) ▪ Fraud detection, other business analytics	**Process an analytical algorithm** over a large set of data in a short period of time, and provide a result users can query or otherwise view. Latency and reliability requirements are less strict than in front-office applications: operations are not time-critical, and analytics can always be re-executed if the process fails.	1. A load process loads data into a database. 2. The messaging server queues an analytic process. 3. An analytic process reads the data and analyzes it. 4. The analytic process writes the result back to the database. 5. The messaging server queues the next analytic process… Until a final result is written to the database. 6. End users query the database for the final results (or alternatively, a reporting system retrieves the results and presents them to end-users).



We see that front-office and back/middle-office analytics systems are both **stateful**—each work unit undergoes a structured workflow—and to enable this workflow, they both use a messaging tier for coordination, a data tier for storage of state information, and a business tier for the processing. In other words, both types of applications are based on a form of the ***tier-based architecture***.[4]

As we shall see, these two similarities are at the root of both applications' scalability problems. In the remainder of this chapter we discuss **stateful, tier-based applications in general**, with the aim of showing that it is precisely these characteristics that make business-critical applications difficult to scale.

2.2. Scalability Barriers Encountered by Tier-Based Applications

As a tier-based application begins to scale out, **latency and data affinity become threatened**, requiring a response from the application's administrators:

Problem ⟶	Scalability Risk ⟶	Typical Fix
Inefficient clustering—expensive communication and data duplication.	Cluster nodes might get overloaded—**latency barrier**.	**Scaling up the nodes**.
Inability to virtualize—each tier cannot expose itself as a single, coherent unit.	Coordination problems between tiers—**consistency barrier**.	Adding a **central coordination component.**
Different clustering models for each tier—causes unnecessary ping-pong inside each tier.	Might overload the tiers—**latency barrier**.	**Scaling up each tier**.
Unknown scaling ratios—for example, if a parsing component is scaled by 50%, the processing component must be scaled by 200%.	Components with high scaling ratio become a bottleneck—**latency barrier**.	**Scaling up the problematic component**—but this only shifts the problem elsewhere.
Messaging overhead—topics/queues grow quickly and processing components receive masses of irrelevant messages.	Processing components might become overloaded—**latency barrier**.	**Scaling up processing components**.
Different H/A models for each tier—H/A and failover is not synchronized across all tiers.	Inability to recover from partial failure—**consistency barrier**.	Adding a **heartbeat system** (or similar mechanism).
Increased physical distribution—components communicate over increasing distances.	The network becomes the bottleneck—**latency barrier**.	**Increasing network bandwidth**.

All the scalability fixes listed above **are only temporary patches**—as the system continues to scale, they need to be applied again and again. They also **make scaling progressively more**

[4] Technologist Scott Burkett provides a formal definition of a tier-based architecture (also known as the n-tier architecture): "An 'n-tier' application architecture is characterized by the functional decomposition of applications, service components, and their distributed deployment. […] A 'tier' itself is nothing more than a functionally separated hardware and software component that performs a specific function." (in *Scaling Your Technology with Your Business*, sourced from http://www.scottburkett.com/index.php/technology/2007-01-08/scaling-your-technology-with-your-business.html).

expensive, because they pour more and more additional resources into the system just to maintain latency and consistency.

The larger the application, the more complex it gets, the more central coordination is required, the more cluster nodes have to work just to deal with the coordination overhead—until eventually, the next capacity unit becomes so expensive that scaling is no longer economically justified. This is the situation we referred to as a **marginal cost barrier**.

Technologist Scott Burkett also writes that tier-based applications tend to hit either crash barriers or a marginal cost barrier as they grow: "Most [tier-based] systems can initially be scaled to a sufficient level […] But that only gets you so far in most cases […] you are either going to hit a ceiling, or spend way too much money to scale it (and even then, there are no guarantees)."[5]

Furthermore, after numerous measures have been applied to solve latency and consistency problems, the system has a large number of 'moving parts'—the clustering model of each tier, the central coordination components, the shared high-availability system. This means that at some point the system will hit a ***feasibility barrier***—it won't be able to scale any further without becoming completely unmanageable, and even ceasing to function.

2.3. Tier-Based == Non-Linear Scalability

Stateful tier-based applications go through a troubling dynamic as they scale up: they constantly hit scalability 'crash barriers'—unacceptable levels of latency and consistency—which can only be avoided by adding central coordination mechanisms and injecting an expensive dose of resources.

The application's administrators find themselves caught between a rock and a hard place: if they choose to apply the scalability fixes, the application grows more and more complex, until it hits a **marginal cost barrier**; but if they don't apply these fixes, the application very quickly hits a **crash barrier**, and must be replaced.

We now see that tier-based applications inevitably hit a scalability barrier—of one sort or another—at some point along the scaling path. Therefore, according to our definition of linear scalability, **tier-based applications *cannot* be linearly scalable**.

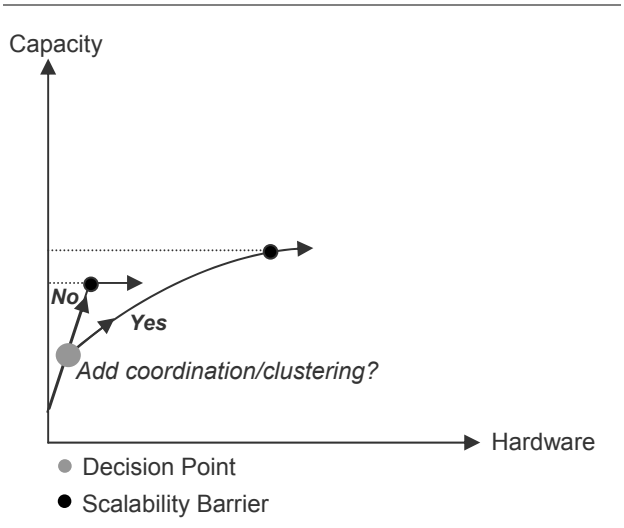


Figure 5 – Scaling Options of Tier-Based Applications

This argument is proven in the general case by the well-known ***Amdahl's Law***,[6] which states the theoretical upper boundary of speed improvements yielded by additional concurrent processors,

[5] Scott Burkett, *Scaling Your Technology with Your Business*, sourced from http://www.scottburkett.com/index.php/technology/2007-01-08/scaling-your-technology-with-your-business.html.

[6] Amdahl, G.M., *Validity of the single-processor approach to achieving large scale computing capabilities*. In AFIPS Conference Proceedings vol. 30 (Atlantic City, N.J., Apr. 18-20). AFIPS Press, Reston, Va., 1967, pp. 483-485.

when there is some non-processing work that all processors must undertake (e.g. synchronization). According to Amdahl's Law, if as little as **10% of computing power is invested in overhead**, then a hundred-fold increase in processing power yields only 9.17 times the speed improvement.[7] Even if an infinite amount of processing power is added, the upper boundary of speed improvement is a factor of ten.

Regardless of the exact percentage of synchronization, remote calls and other overhead in actual tier-based systems, it is obvious that there is some overhead occupying all processors in the system—and in this case, **Amdahl's Law proves that linear scalability is impossible**.

Specifically, front-office applications and back/middle-office analytics applications cannot be linearly scalable—and so, in the long run, there is **no way to guarantee these applications' survival**.

2.4. The Architecture is the Bottleneck

At different points along the scaling path, different components can become the bottleneck—the messaging server, the transaction coordinator, the database, the business logic units, the network. But these **specific components are not the cause of the problem**—problems arise in the interfaces between the tiers, and due to their increasingly complex interaction.

Because the system is divided into separate tiers, it increases in complexity as it scales, requires more and more overhead just to manage this complexity, and makes it more and more costly to increase capacity. **The tier-based model is the real bottleneck**—and the only way to release this bottleneck is a change of architecture.

[7] Calculation appears in R.J. Lorimer, *Performance, Understanding Amdahl's Law*, sourced from http://www.javalobby.org/java/forums/t84101.html.

3. Towards a Linearly-Scalable Architecture

We have shown that application scalability pains are due to a non-scalable architecture. If so, is there an alternative architecture that **is** linearly scalable, even in low-latency applications, which are notoriously difficult to scale?

Section 3.1 shows that the key to a linearly scalable architecture is to **expand the application by adding self-sufficient units**. This way, as the application grows, there is no complex coordination that consumes resources and leads to scalability barriers.

Section 3.2 explains how different units, although they are self-sufficient, can **share state to ensure data affinity**. This is solved by the concept of a *processing unit*, which compresses all steps of the business process into a single machine.

Section 3.3 discusses the importance of simplicity: why a linearly scalable application must **look and behave as one server** as it scales up, presenting a single front to designers, developers and clients.

3.1. Linear Scalability through Self-Sufficient Units

The natural way to achieve linear scalability is hinted in our definition:

> ***Linear scalability*** occurs when each new hardware unit always contributes the same amount of additional capacity.

More concretely, each time you buy a new machine and add it to the system, the application's capacity grows by **X** (a constant number); if you buy three more machines, your capacity grows to **4X**.

To achieve this, all you need to do is **make each machine self-sufficient**. Say one machine is capable of processing 1000 transactions per second (X=1000), and each machine is completely self-sufficient: able to process user requests completely on its own, without consulting any external resource. Adding one such machine adds 1000 transactions per second (X); adding three machines adds 3,000 transactions per second, for a total of 4000 (4X). So you have linear scalability.

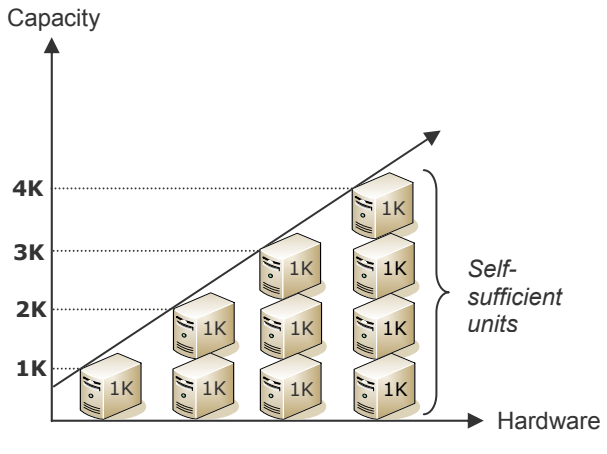


Figure 6 – Scaling Up by Adding Self-Sufficient Units

We are coming to a realization that **self-sufficiency is a sufficient condition for linear scalability**—make each hardware unit self-sufficient, and you have linear scalability.

This is a clear departure from the tier-based architecture: instead of running each application tier in a separate cluster of computing resources, all the tiers are compressed into a single unit, which is

duplicated when the application is scaled. **The middleware problem**—the frightening complexity of wiring different components together, which gets worse and worse as the application scales—**simply evaporates** when each component becomes self-sufficient.

In a stateless environment, self-sufficiency is quite easy to achieve. But in a stateful environment, in which different components perform different parts of a structured workflow, a paradox arises: how can you manage a workflow when each machine is completely self-sufficient? How can machines share state information between them?

3.2. Self-Sufficiency with Workflow Management: the Processing Unit

The key to resolving the paradox is *collocating* all steps of the business process—putting them on the same machine, in the same JVM. This requires developing a software component that can accept a user request, **perform all steps of the transaction on its own**, and provide a result. This is a *processing unit*—a mini-application which can perform the entire business process for a small number of user requests (as many as can be handled by a single machine).

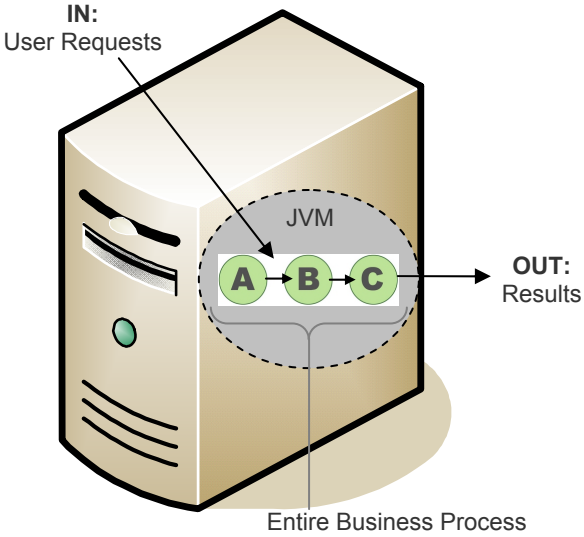


Figure 7 – A Processing Unit

The idea of a processing unit removes the need for sharing of state information and partial results between different components of the application running on different physical machines. Information only needs to be shared **within the processing unit**—in other words, within the confines of the local machine. So each processing unit can manage its own workflow.

The processing unit does not have to be one monolithic unit—it can be composed of several subcomponents, which can even be defined as services participating in an SOA. The only requirement is that all these subcomponents be collocated in the same JVM.

Distributed Messaging and Data Storage

Since processing units are self-sufficient, it is only natural that each processing unit provide messaging and data storage facilities to its own service instances. This has the immediate advantage of **saving the network calls and overhead** incurred in connecting to a remote database.

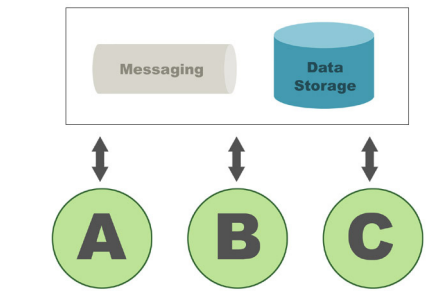


Figure 8 –Processing Unit with Independent Messaging and Data Storage

But not only do the services save the remote call—they are also spared the I/O overhead of saving the data to disk. Recall that all the services are collocated on the same machine, so all the information they share between them **can be stored in local memory**.

This means that within the processing unit, communication and coordination can take place with the **lowest possible latency**, that of in-memory access.

In practical terms, this means that instead of accessing a messaging queue on an external server, services access a messaging queue in local memory. When work has been done, instead of connecting to a central database and storing partial results there, each service stores its results in local memory, and can read data it needs from local memory, where it has been placed by other services.

What About Reliability?

When everything is stored in local memory, the question of reliability arises—what happens if the machine running the processing unit fails? Will all state information be lost? The simple answer is no: it is easy to guarantee the reliability of each processing unit's data—much easier than in tier-based applications.

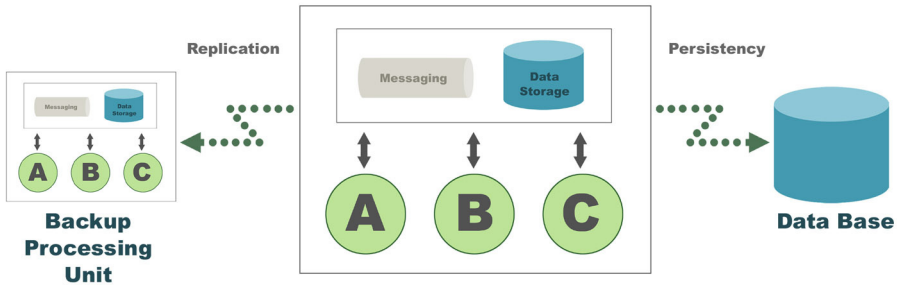


Figure 9 – Reliability Options: Persistency or Replication

One way to do this is to run two identical processing units on two different machines, **one as primary, the other as backup**. Messages and data can be replicated between these two processing units asynchronously (i.e. once in a while), without affecting latency. If reliability requirements are particularly strict, the replication can be made synchronous, at a certain latency cost.

This not only safeguards against failure of the local machine—it also enables **hot failover** between the two processing units. The backup shares the state data stored in the primary, so it can take over at the precise point of interruption, complete the work and return a result—without users ever finding out about the failure.

Another way to safeguard the data is to **persist it to a database**. (This can be a local database, collocated on the same machine as the processing unit, which makes persistency very inexpensive.) This way, data in the processing survives a restart. To safeguard against a hard-drive crash, the data can also be persisted, either synchronously or asynchronously, to a remote database.

The point is that the processing unit **can be made as reliable as needed**. In a processing unit some data can be stored in memory only; other data can be stored in memory and replicated to the memory of another machine; while particularly sensitive data can even be stored to a remote database.

This is a vast improvement over the tier-based model, which forces all processing components to store data to a database, and pay the high price of persistency, even if this type of reliability is not really required.

Partitioning the Application

In the processing unit model, the application is partitioned into a number of mini-applications, each of which serves a portion of the users.

To make this possible, a **partitioning mechanism** is needed to distribute user requests between the processing units.

The best way to do this is by **content-based routing**. The partitioning mechanism needs to be able to read the content of the user request, and use one or more fields of the content as the partitioning criterion (this criterion needs to be selected so that all the user requests that relate to each other or share the same data go to the same processing unit).

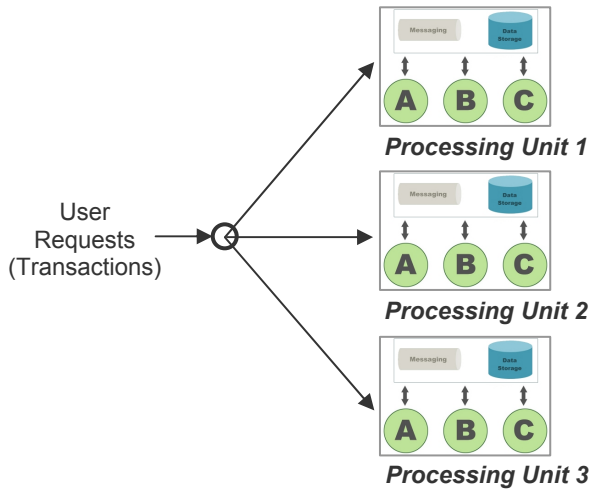


Figure 10 – Partitioning the Application

This kind of routing is **very inexpensive because it is a 'no brainer'**—if the request has value A it goes to Processing Unit 1; if it has value B it goes to Processing Unit 2—there is no need for communication with or between the processing units, no need to keep track of which requests were sent where, and no need to be aware of the current status of each request.

Most importantly, content-based routing **does not increase in complexity as the application scales**, ensuring that scalability remains linear.

3.3. Simplicity—Making the Distributed Application Behave and Look as One

The concept of a processing unit makes linear scalability possible, but it is not enough that the application can scale indefinitely. A critical requirement is that the application should not increase in complexity as it grows:

- **From the designer's perspective**, there should be no scalability considerations—the designer should have full flexibility in constructing the system, without having to foresee or take into account its future growth.
- **From the developer's perspective**—the application should appear as one coherent unit, presenting a unified programming model and a single API, even as it grows. It should not be necessary to integrate new products or technologies to scale the application in the future.
- **From the administrator's perspective**—the administrator should be required only to define general software and hardware requirements, and required service levels; as more hardware resources are added, the application should dynamically adapt to the new topology. Deployment should always be as simple as running the application on a single machine.
- **From the client's perspective**—the client should see the entire application as a single server, and be able to subscribe to queues, perform queries, etc., using a single operation.

4. SBA: Scalability as Open Road

In the previous chapter we introduced the concept of a self-sufficient *processing unit*, as a solution to the scalability barriers faced by tier-based applications. Space-Based Architecture (SBA) is a way to implement processing units in your application, transforming its scalability from dead end to open road:

4.1. The Space: the Beating Heart of a Processing Unit

SBA is based on a middleware infrastructure called the *space*. The space is a distributed shared memory unit, which facilitates **both data caching and messaging using a single technology**, a single API and a single clustering model.

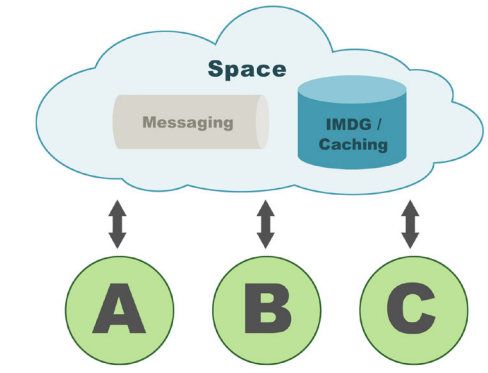


Figure 11 – An SBA Processing Unit

The space sits at the core of the processing unit—like a conveyor belt in a production line, it helps coordinate different stages of the business process, ensuring transactional security; and allows each stage to physically deposit its partial results on a shared work area, which can be accessed by other processing stages in turn.

When you transition your application to SBA, each component of the business process becomes a service, and all the services required to process a transaction, from start to finish, are deployed inside the processing unit. The processing unit thus comprises a set of business logic services and a middleware component—the space.

High Availability and Hot Failover

The space has a built-in clustering model, which allows it to share data and messages with other spaces on the network, without requiring a central server.

The first implication is that the process data and messages in the space **can be made highly available**, via replication between spaces running on different physical machines. Special backup spaces can be maintained (without the business process services), or alternatively, entire processing units can serve as live backups of each other.

Because different machines are running the same processing units, this enables **hot failover**—a transaction can be replicated to another space in an identical processing unit, and thus can be immediately resumed from the place it was interrupted.

Each space's data can also be **persisted to a database**; this occurs in the background, without affecting the latency of the processing unit.

Distributed Content-Based Routing

As we discussed earlier, a mechanism is needed to route each client request to the appropriate processing unit. This too is made possible by the space's clustering capabilities.

Clients of the application **execute requests against a generic cluster proxy**, without knowing which processing unit is relevant for their request, or even which processing units exist. The cluster matches the request at the object level, and routes it to the appropriate space.

This is **not done by a central server**, which might get overloaded—rather, the routing is performed by the individual spaces, in a manner which is completely transparent to the client.

No-Overhead Messaging

In traditional messaging, workflow components receive a large volume of messages in the queue they subscribe to. This generates overhead, because each component needs to work hard to screen out irrelevant messages.

By contrast, the space allows **direct in-memory access to specific messages**—each business process service can match on the messages it needs. Thus, there is no need to invest resources in screening out unneeded messages, and no messaging-related overhead.

Low-Latency Data Access

The space not only facilitates messaging, but also acts as a virtual in-memory data store for the business process services.

Because the space is collocated with these services, they can save results to the virtual data store, and retrieve results from it, **at in-memory speeds**. The overhead associated with maintaining database sessions and communicating with a remote database is totally eliminated—unless there is a special need to persist data to a central database, and even then the space can do this asynchronously, maintaining minimal latency.

4.2. Deploying the Processing Unit

A central part of SBA is a deployment framework that takes care of running the processing unit on physical machines. The framework consists of *SLA-driven containers* that run on all the computers in the deployment scope.

A *deployment descriptor* is defined, which states which services each processing unit should comprise, their mutual dependencies, the software and hardware resources they require, and the SLA they are required to meet. The deployment framework then goes into action, **automatically deploying the processing unit** on as many physical machines as necessary to meet the SLA.

The deployment framework performs **distributed dependency injection**—each time an instance of the processing unit is launched, the framework provides each service within the unit with a handle to all the other services it needs to work with.

This means that services do not need to discover each other and check mutual availability. The deployment framework sees the entire deployment scope, and knows which services are currently available. A processing unit is **only deployed if all its dependencies can be currently fulfilled**.

From Ten Machines to Ten Thousand

Via the user interface of the deployment framework, it is easy to deploy one or more processing units on as many machines as required—with one click. Deployment doesn't become even a bit more complex when scaling from ten machines to ten thousand; the deployment descriptor describes a generic scaling policy for all possible deployment sizes.

Scale Up, Scale Out, or Both

Today there are two major routes to scaling hardware, usually thought of as mutually exclusive: *scaling up*—adding more CPUs to a multi-core machine, and *scaling out*—adding more machines, using a clustering model.

In SBA, these two models are not mutually exclusive, but complementary. There is **no need to make a design-time decision** on which scaling model to adopt.[8] At any point during the application's lifetime, machines can be scaled up, and the existing processing units can accept increased load; or additional machines can be added, and additional processing units can be deployed on them.

This **two-dimensional scaling** provides great flexibility in selecting hardware resources for expansion—if today scaling out provides the most cost-effective solution, machines can be added; when the capacity of individual machines grows, and scaling up becomes more attractive, machines can easily be scaled up, without changing the application.

Another advantage is that the same application can be supplied to customers with different deployment scenarios. This can represent a significant saving on development costs, because there is no need to custom-build an application according to the customer's scaling requirements, or provide different flavors for each possible deployment scenario.

Dynamic Scalability

The deployment framework dynamically responds to the application workload: if service levels fall below the SLA, **additional processing units are automatically added**; when workload decreases, processing units are shut down, freeing up computing resources.

Because processing units can run on just about any machine, it is easy to expand the pool of computing resources by **adding commodity hardware**, such as Blade servers.

---

[8] Technologist Scott Burkett describes the current situation, in which scaling up or out **is** a design-time decision: "Horizontal scaling [=scaling out] only makes sense if the service you are attempting to scale was designed to be extended in this manner. […] if you are designing the software, you will want to take this into account as you build it." (in *Scaling Your Technology with Your Business*, sourced from http://www.scottburkett.com/index.php/technology/2007-01-08/scaling-your-technology-with-your-business.html).

4.3. Simplicity Achieved—the Application Behaves and Looks as One Server

A true-to-form implementation of SBA achieves true *simplicity*—the distributed application behaves as one coherent unit, as if it was a single server. Simplicity is achieved in several dimensions:

- **From the designer's perspective**—scalability is not a design consideration. The designer constructs a self-contained processing unit that can be duplicated as many times as needed to facilitate future scaling.
- **From the developer's perspective**—the system presents a unified API and clustering model. Developers continue to write code as if for a single server; the physical distribution of the runtime environment is completely abstracted through the middleware implementation and the Spring integration. Efforts can focus on a standalone processing unit, in a standalone application context; this also enables testing 90% of the business logic on a single machine, without running a cluster.

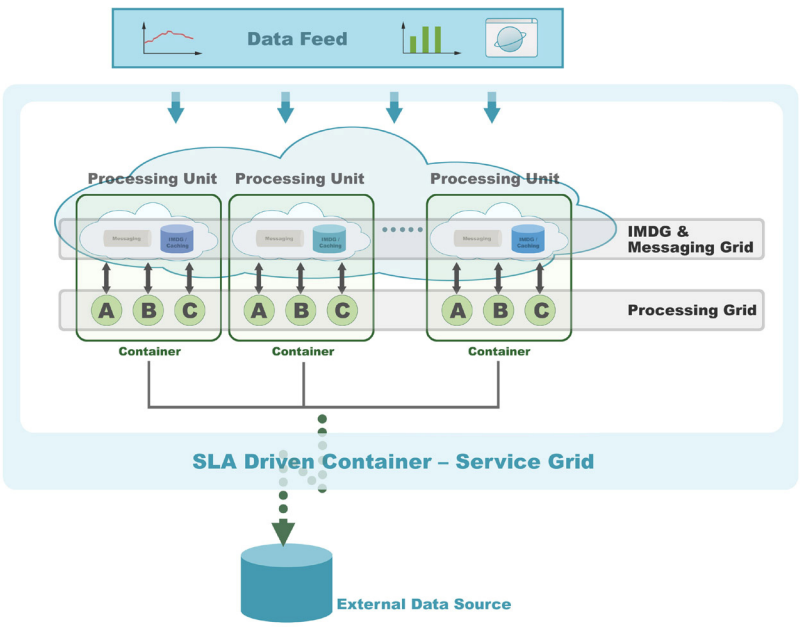


Figure 12 - SBA Application Looking and Behaving as One Server

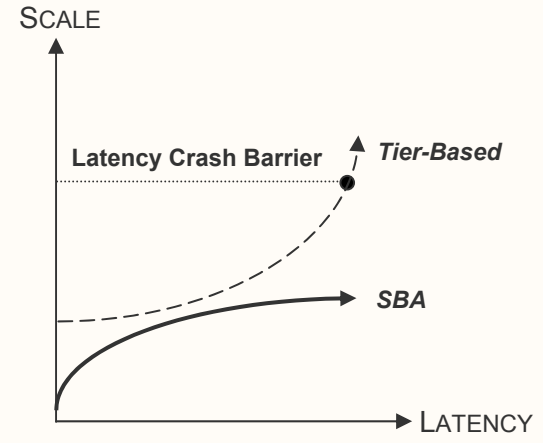
- **From the administrator's perspective**—the entire application is deployed through a single deployment command, as if it was running on a single server. The deployment command is SLA-driven, depending only on the predefined hardware and software requirements of each processing unit—the physical topology and machine availability is totally abstracted from the administrator.
- **From the client's perspective**—clients see the entire cluster of processing units as if it were a single server. They can subscribe to messages, perform queries, or even read data directly from the space of a processing unit in a single operation, without being aware of the underlying topology. Operations are maintained transparently by the SBA middleware.

The bottom line is that **the application's scale becomes transparent** to all involved, as if it was a single server. Complexity, overhead and cost—the three biggest challenges of a stateful application as it scales up—stay exactly the same as the distributed application grows to the sky.

5. Conclusion: The SBA Value Proposition

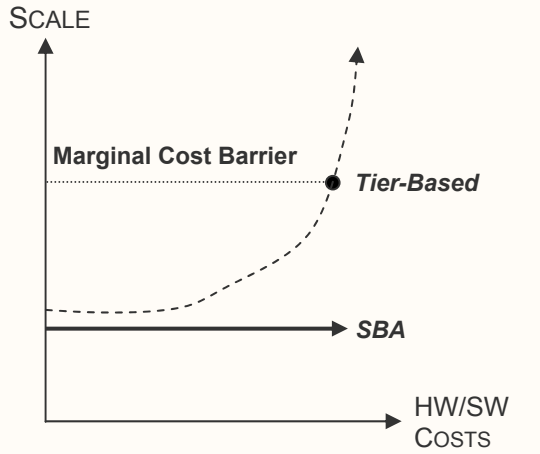
SBA's primary value proposition is that it guarantees the long-term survival of today's business-critical applications, by removing the imminent threat of scalability barriers. In so doing, it improves the scalability of tier-based applications in four dimensions: latency, hardware/software costs, development costs, and data affinity.

Scaling with Consistent Low Latency



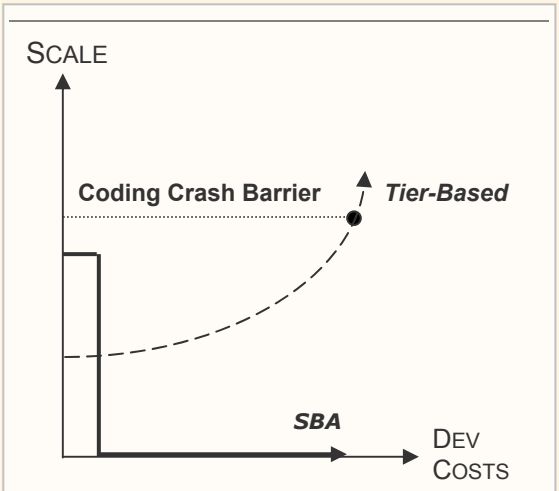
SBA applications maintain minimum possible latency throughout the scaling path, avoiding the latency crash barrier.

Scaling with Fixed Resource Costs



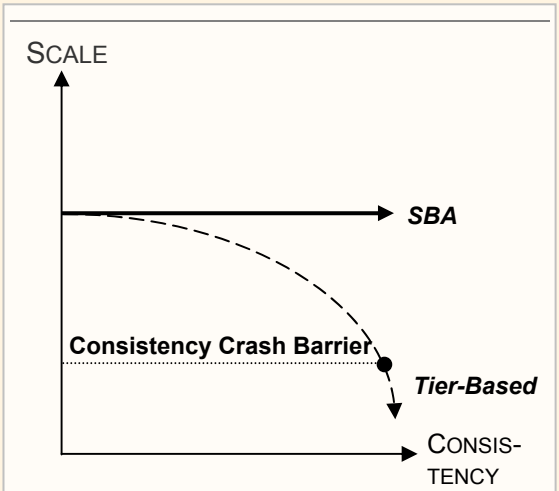
SBA provides a predictable, fixed cost per additional hardware unit, including its associated software licenses.

Scaling without More Development



SBA applications require a development effort at first, but thereafter scale indefinitely without any changes to architecture or implementation.

Scaling with Guaranteed Data Affinity



SBA applications provide high data affinity—even at the level required by front-office applications—and guarantee consistency at any scale.

APPENDIX: Implementation Example—A Trading Application

This appendix shows how SBA would be implemented for a simple distributed application: a trading system that accepts buy and sell orders from clients. This application is transactional (there are two sequential steps in the business process), and requires very low latency, so that buy and sell orders can be executed in real time.

To see this implementation example, click here.

U.S. Headquarters
GigaSpaces Technologies Inc.
1250 Broadway, Suite 2301
New York, NY 10001
Tel: 646-421-2830
Fax: 646-421-2859

U.S. West Coast Office
GigaSpaces Technologies Inc.
555 California St., 3rd Floor
San Francisco, CA 94104
Tel: 415-568-2125
Fax: 415-651-8801

Europe Office
30 Borough High Street
London SE1 1XX
United Kingdom
Tel: +44 709 286 3096
Fax: +44 709 286 3097

International Office
GigaSpaces Technologies Ltd.
4 Maskit Street, P.O. Box 4063
Herzliya, 46140, Israel
Tel: +972-9-952-6751
Fax: +972-9-957-6780