NoCAP

Or, Achieving Scalability Without Compromising on Consistency

May 2011

Introduction

As with any new movement, NoSQL has brought with it a cycle of hype, which at its peak leads to adoption of technologies without a great deal of questioning. A primary goal (and selling point) of NoSQL systems is scalability enhancement, resting on the premise that the requisite strong network partition tolerance compels giving up either consistency or availability – an idea based on the CAP theorem.

CAP is often associated with scalability, without a full understanding of its implications – or the alternative approaches available. Hence, I've coined the term ***NoCAP*** – meant to illustrate that you can achieve scalability without compromising consistency… Or at least, not to the extent that many disk-based NoSQL implementations impose.

Executive Summary

According to the CAP theorem, it is impossible for a distributed system to have all three CAP properties – consistency (C), availability (A), and partition tolerance (P) – necessitating a choice of only two: Some suggest choosing AP and compromising on consistency. Others suggest CA as a better set of tradeoffs.

This paper presents the argument that it is not necessary to completely give up partition tolerance by choosing CA, or consistency by choosing AP. Instead of viewing each CAP property in absolute terms and selecting only two, we can adopt a more relaxed approach that applies various degrees of *all three*, and compromise on the degree in which we apply each property based on the application's business requirements. In other words, address the most likely failure and network partition scenarios, and compromise only in areas where they are less likely to occur.

A common GigaSpaces clustering topologies is used as a reference for this model, with a detailed illustration of how the topology applies to all three CAP properties.

Recap on CAP

The **CAP theorem**, also known as **Brewer's theorem**, states that it is impossible for a distributed computer system to simultaneously provide all three of the following core system requirements:

- *Consistency* (all nodes see the same data at the same time)
- *Availability* (node failures do not prevent survivors from continuing to operate)
- *Partition Tolerance* (the system continues to operate despite arbitrary message loss)

(Wikipedia)

Is this truly our only choice?

This paper is intended to provide an aggregated view of the different perspectives on this question and plant the seeds for an alternative approach.

CAP and NoSQL

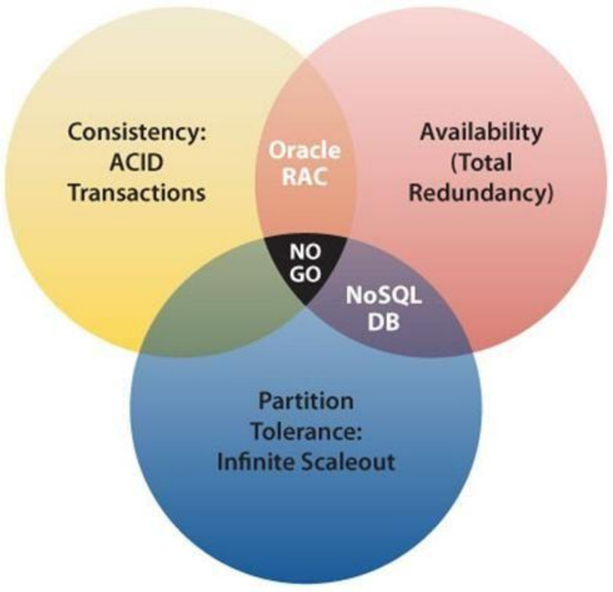
Many of the disk-based NoSQL implementations originated from the need to deal with write-scalability. This was largely due to changes in traffic behavior that were a result of social networking, in which most of the content is generated by the users and not by the site owner.

In a traditional database approach, achieving data consistency requires synchronous write to disk and distributed transactions (known as the ACID properties). It was clear that the demand for write-scalability would conflict with the traditional approaches for achieving consistency (synchronous write to a central disk and distributed transactions).

The apparent solution to this challenge included:

1. Breaking the centralized disk access through partitioning of the data into distributed nodes.
2. Achieve high availability through redundancy (replication of the data into multiple nodes)
3. Use asynchronous replication to reduce the write latency.

The assumptions behind point 3 are at the heart of this paper.


Consistency: ACID Transactions
Oracle RAC
Availability (Total Redundancy)
NO GO
NoSQL DB
Partition Tolerance: Infinite Scaleout

Graphical representation of the CAP Theorem (Source)

The Consistency Challenge

A common assumption behind many NoSQL implementations is that write-scalability requires pushing as many operations as possible from the write-path to a background process, to minimize the time a user transaction is blocked on write. The implication is that asynchronous write reduces consistency between write and read operations (read operations can return older versions than write operations).

Various algorithms have been developed to address this type of inconsistency challenge, which is often referred to as Eventual Consistency.[1]

[1] For more information, see *Jeremiah Peschka's post, Consistency Models in Non-relational DBs*, providing a summary of the CAP theorem, Eventual Consistency model and other related principles (such as **BASE** – Basically Available Soft-state Eventually, NRW, and Vector clock).

Do We Really Need Eventual Consistency to Achieve Write Scalability?

A quick introduction to the term “scalability”, which is often used interchangeably with throughput:

The terms “performance” and “scalability” are commonly used interchangeably, but the two are distinct: performance measures the speed with which a single request can be executed, while scalability measures the ability of a request to maintain its performance under increasing load

~Steve Haines

In our specific case, this means that write scalability can be delivered primarily through point 1 and 2 above (Break the centralized disk access through partitioning of the data into distributed nodes, and Achieve high availability through redundancy and replication of the data into multiple nodes), where point 3 (Use asynchronous replication to those replicas to avoid the replication overhead on write) relates to write throughput and latency, and *not* scalability. Thus,

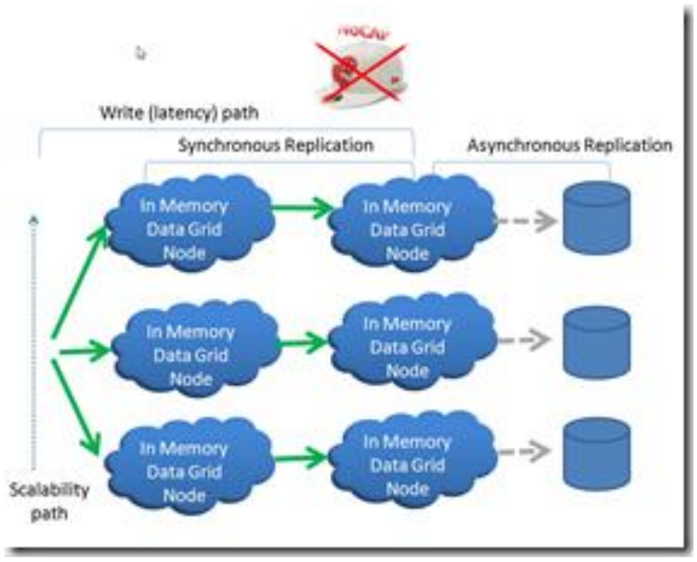
Eventual consistency has little or no direct impact on write scalability.

Specifically, it is often enough to break the data model into partitions (shards) and break away from the centralized disk model to achieve write scalability, and thus sufficient throughput and latency levels.

The use of asynchronous write algorithms to optimize write performance and latency should be considered, but due to its inherent complexity, consider it only after trying simpler alternatives, such as using db-shards, FLASH disks, or memory-based devices.

Achieving Write Throughput Without Compromising Consistency or Scalability

This diagram illustrates one way that write scalability and throughput can be achieved without compromising consistency.



Data is broken into partitions to handle write scaling between nodes. To achieve high throughput, in-memory storage is used instead of disk. Because in-memory devices tend to be significantly faster and more concurrent than disk, and because network speed is no longer a bottleneck, high throughput and low latency can be achieved even using synchronous write to the replica.

The only place that uses asynchronous write is the write to long-term-storage (disk). Because the user transaction doesn’t access long-term storage directly through the read or write path, it isn’t exposed to the potential inconsistency between memory storage and the long-term storage. Long-term storage can be any disk-based alternative, from a standard SQL database to any of the disk-based NoSQL engines.

The other benefit of this approach is that it is significantly simpler. Simpler not just in terms of development, but simpler to maintain compared with the Eventual Consistency alternatives. For distributed systems, simplicity often correlates with reliability and deterministic behavior.

Availability vs. Partition Tolerance, and CAP Consideration Arguments

There is often a lack of clarity in the definitions of Availability and Partition Tolerance, and the difference between them.

The original definitions by Gilbert and Lynch:

Availability
For a distributed system to be continuously available, every request received by a non-failing node in the system must result in a response. That is, any algorithm used by the service must eventually terminate... [When] qualified by the need for partition tolerance, this can be seen as a strong definition of availability: even when severe network failures occur, every request must terminate.

Partition tolerance
In order to model partition tolerance, the network will be allowed to lose arbitrarily many messages sent from one node to another. When a network is partitioned, all messages sent from nodes in one component of the partition to nodes in another component are lost. (And any pattern of message loss can be modeled as a temporary partition separating the communicating nodes at the exact instant the message is lost.)

In his post Errors in Database Systems, Eventual Consistency, and the CAP Theorem, **Dr. Michael Stonebraker** argues that because partition failures are rare, you might sacrifice partition tolerance for consistency and availability:

> *Obviously, one should write software that can deal with load spikes without failing; for example, by shedding load or operating in a degraded mode. Also, good monitoring software will help identify such problems early, since the real solution is to add more capacity. Lastly, self-reconfiguring software that can absorb additional resources quickly is obviously a good idea.*
>
> *In summary, one should not throw out the C [consistency] so quickly, since there are real error scenarios where CAP does not apply and it seems like a bad tradeoff in many other situations.*

Henry Robinson responded that failures are inevitable, and therefore partition tolerance is mandatory:

> *Partition tolerance is not something we have a choice about designing into our systems. If you have a partition in your network, you lose either consistency (because you allow updates to both sides of the partition) or you lose availability (because you detect the error and shutdown the system until the error condition is resolved). Partition tolerance means simply developing a coping strategy by choosing which of the other system properties to drop. This is the real lesson of the CAP theorem – if you have a network that may drop messages, then you cannot have both availability and consistency, you must choose one. (Problems with 'Partition Tolerance')*

Coda Hale also argues that you can't give up partition tolerance in *his* response, You Can't Sacrifice Partition Tolerance. Hale suggests a more relaxed version of availability through graceful degradation, based on the yield and harvest model from Brewer's ACM article, Lessons from Giant-Scale Services:

> ***You cannot choose both consistency and availability in a distributed system.***
>
> *Of the CAP theorem's Consistency, Availability, and Partition Tolerance, Partition Tolerance is mandatory in distributed systems. You cannot* ***not*** *choose it. Instead of CAP, you should think about your availability in terms of* yield *(percent of requests answered successfully) and harvest (percent of required data actually included in the responses) and which of these two your system will sacrifice when failures happen.*

Stonebraker: Network partition and machine failure are two separate things

Both Robinson and Hale referred to machine failure as part of their argument that you can't do without partition tolerance. Stonebraker provides clarification on this point in his response to Hale:

> *.., the dead node is in one partition and the remaining N-1 nodes are in the other one. The guidance from the CAP theorem is that you must choose either A or C, when a network partition is present. As is obvious in the real world, it is possible to achieve both C and A in this failure mode. You simply failover to a replica in a transactionally consistent way. Notably, at least Tandem and Vertica have been doing exactly this for years. Therefore, considering a node failure as a partition results in an obviously inappropriate CAP theorem conclusion.*

Daniel Abadi outlines four issues in the current definition of CAP in Problems with CAP, and Yahoo's little known NoSQL System:

- *The definition of CP looks a little strange --- "consistent and tolerant of network partitions, but not available" --- the way that this is written makes it look like such as system is never available --- a clearly useless system*
- *The roles of the A and C in CAP are asymmetric. Systems that sacrifice consistency (AP systems) tend to do so all the time, not just when there is a network partition*
- *What if there is a network partition? What does "not tolerant" mean? In practice, it means that they lose availability if there is a partition. Hence CP and CA are essentially identical*
- *Lack of latency considerations in CAP significantly reduces its utility*.

Abadi suggests an alternative definition to CAP:

> *...CAP should really be PACELC --- if there is a partition (P) how does the system trade-off between availability and consistency (A and C); else (E) when the system is running as normal in the absence of partitions, how does the system trade-off between latency (L) and consistency (C)?*

Conclusions on CAP Property Considerations

A major issue with CAP's current definition is that it describes each of the CAP properties in absolute terms. However, each property can be applied in various degrees. Brewer's definition of yield and harvest is one example of a more relaxed version of availability. Interestingly, Gilbert and Lynch also recognize that it might be possible to achieve tradeoffs that provide both Availability and Consistency:

> *…in partially synchronous models it is possible to achieve a practical compromise between consistency and availability. In particular, most real-world systems today are forced to settle with returning "most of the data, most of the time." Formalizing this idea and studying algorithms for achieving it is an interesting subject for future theoretical research.*

The same applies to Partition Tolerance: There are various degrees of partition tolerance. Probably the most extreme examples relate to WAN, but as Stonebraker argues, even that is not necessarily a high-likelihood scenario for most applications:

> *There is enough redundancy engineered into today's WANs that a partition is quite rare. My experience is that local failures and application errors are way more likely. Moreover, the most likely WAN failure is to separate a small portion of the network from the majority. In this case, the majority can continue with straightforward algorithms, and only the small portion must block. Hence, it seems unwise to give up consistency all the time in exchange for availability of a small subset of the nodes in a fairly rare scenario.*

So, the question of whether or not you can address partition tolerance shouldn't be measured in absolute terms but against the most likely scenario for your application. The answer as to what is a most likely partition scenario really comes down to common sense and can vary with the passage of time and changing application business needs.

The emergence of the cloud might also change some of our core assumptions about how we deal with failure. With the cloud, it is easy to bring up an alternate machine to deal with a failure in a matter of minutes, and it is a fair assumption that the underlying infrastructure is robust, with plenty of built-in redundancy. That in itself changes the possible failure scenarios, as well as how to deal with them, compared to the time when the original CAP Theorem was written (about a decade ago).

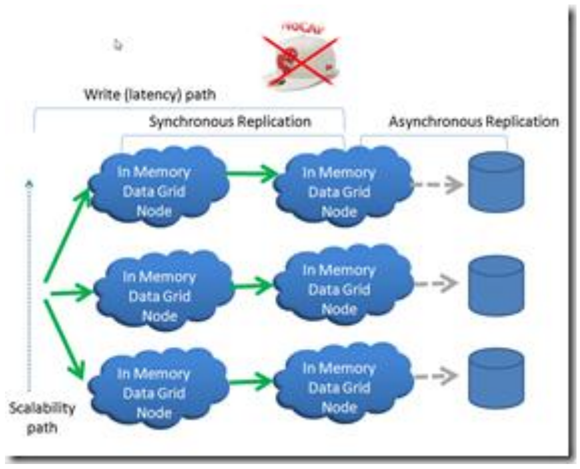
Another source of confusion is the use of Amazon, Google, and Facebook as references to justify the eventual consistency model. What is often forgotten is that Amazon, Facebook, and Google face rather unique challenges, and even those three still rely on strongly consistent systems for the majority of their applications. As **Derrick Harris** notes in Will Scalable Data Stores Make NoSQL a Non-Starter?:

> *Facebook famously invented the NoSQL Cassandra database but still relies on the venerable MySQL-plus-memcached combination for the bulk of its critical operations.*

The bottom line is that giving up consistency should be a last resort. Rather than giving up Consistency for Partition Tolerance, consider a different set of tradeoffs that offers varying degrees of Consistency, Availability, and Partition Tolerance that fit specific business needs, and which also take performance and latency tradeoffs into account.

GigaSpaces Clustering and CAP Properties

Now, let's take a look at how the GigaSpaces clustering topology introduced earlier applies to the idea that all three CAP properties can coexist if you do not adopt an absolute approach to each, but compromise only on the least likely failure scenarios.



Consistency

- **Consistency under concurrent updates.** To ensure consistency with concurrent updates on the same data record, each individual record is mapped to a single logical partition at every given point in time. To ensure scalability, different records of the same logical table are written to multiple partitions in parallel, as illustrated in the diagram. Each partition supports the various locking semantics (pessimistic, optimistic (versioning), dirty-read…) to control concurrent access of the same record within the context of a single partition.
- **Consistency between two or more replicas.** To ensure continuous high availability, one or more copies of the data are made. Asynchronous replication can result in read and write operations

hitting two nodes simultaneously, thus reading two different versions of the same data. While some algorithms are meant to handle this situation, our model avoids the situation entirely through synchronous backup. The performance overhead of the synchronous replication is fixed and not proportional to the cluster size (each partition replicates data only to its backup replica). Replication to the database is kept asynchronous to reduce the overhead of writing to disk.

- **Transaction consistency.** Single operations or groups of operations can be executed under transactions, ensuring ACID properties. Transactions can be made local to each partition, in which case they are bound to the scope of a single partition and highly optimized in terms of performance. Transactions can also span nodes (clearly, resulting in higher overhead).
- **Ordering**. All operations are ordered based on the time they were written. This is specifically relevant to ensure the consistency between the in-memory cluster and the long-term storage which is being updated asynchronously.

Availability

- **Primary failure.** We always keep one or more replica node as a hot backup. The hot backup immediately takes over if a primary node fails. The hot backup nodes uses synchronous replication to ensure zero data loss before fail-over takes place.
- **Backup failure.** When a backup node fails, the primary continues to serve requests and log the operation in a redo log. In parallel, a new backup is provisioned on demand to take over from the failed one. This involves both a provisioning process (in which a new backup is created) and a recovery process (during which the backup gets its state).
- **Multiple node failure.** To increase availability, some NoSQL variants suggest three or more replicas per partition to handle multiple node failure. This obviously comes with huge overhead – each terabyte of data results in two terabytes of redundant information for backup purposes, and – equally important – with the continual overhead of keeping all of the replicas up to date.

  An alternative approach is to use **on-demand backups**, which are provisioned automatically as soon as a node fails. If spare capacity exists within the current machine pool, the backup is provisioned onto an alternate machine within the existing pool, which can take as little as a few seconds (depending on the amount of data per partition). If no machine is available, a completely new machine is started, and a new backup is provisioned into that machine. This process can take a few minutes. As soon the node starts, it uses a primary election protocol to find the master node within its group and only then it boots up. The startup process includes a recovery stage in which the node recovers its state from either the master node or the available replicas. The source node also stores all the updates since the recovery started in a redo log and replays all updates to fill in the gap created from the time the node started its recovery process.
- **Client failure.** Clients use a cluster-aware proxy to communicate with the cluster. The smart proxy ensures that write or read operations are always routed to one of the available partitions. The routing occurs implicitly, so the client is not exposed to fail-over situations.
- **Discovery protocol.** The GigaSpaces cluster discovery mechanism is based on the Jini specification. Services use the discovery protocol to find nodes within the cluster and share cluster state amongst all nodes.

Partition Tolerance

- **Network partition between primary and backup.** When the connection between two nodes fails, the primary node logs all the transactions into a FIFO queue, or redo log. As soon as communication is reestablished, all the data is replayed to the backup. If the backup fails completely, the system starts a new instance as previously described.
- **Network partition and long-term persistency.** If communications with the long-term persistency data store fail, the replica logs all operations until the connection is reestablished. The log is also

replicated to a backup node to ensure that the data won't be lost in case the primary partition fails before the data was successfully committed to the long-term storage.

- **Network partition between two or more data center sites.** The usual reference scenario for multi-site partition tolerance is two separate locations that continue to work independently. It is important to note, however, that there are two classes of multi-site deployment that differ fundamentally regarding network partition:
  - **Disaster recovery site:** Often located in close geographical proximity to the primary site and is backed by high bandwidth and a redundant network.
  - **Geographically distributed sites over internet WAN**: Multiple sites spread globally. Unlike the disaster recovery site, these tend to operate under lower SLAs and significantly higher latency.

  DEALING WITH DISASTER RECOVERY SITE PARTITION:

  Disaster recovery sites are very much like any node in a local network, but often live in different network segments and with higher latency than local networks. Nodes within a cluster can be tagged with a zones tag to mark their data-center affinity. The system can use this information to automatically provision primary and backup nodes between the two sites. It uses the zone tag to ensure that primary and backup are always spread between two data centers.

  - **Consistency & Availability** – The consistency and availability mode are the same as the LAN-based deployment as described previously, but the performance and throughput per partition would be lower due to the higher latency associated with synchronous replication.
  - **Partition Tolerance** – The system continues to function even in a case where entire site fails or become disconnected. The system will continue to work through the available site. The system will also rebalance itself as soon as the communication between both sites is reestablished in order that the load will be evenly distributed between the sites.

  DEALING WITH GEOGRAPHICALLY-DISTRIBUTED PARTITION:

  Because nodes are spread over internet connections where SLAs are lower and latency is much higher, it is impractical to treat all the nodes as a single cluster (as in the previous case). It is more practical to use a federated cluster deployment (a cluster of clusters), using asynchronous replication to synchronize the multiple sites, and thereby avoid the extreme latency overhead. In this mode, all three CAP properties cannot be achieved, and AP is commonly chosen over CA. However, choosing AP does not necessarily mean completely giving up consistency. Here is an architecture that can provide a reasonable degree of consistency with a slight compromise on absolute availability and partition tolerance:

  - **Consistency:** Each site is the sole owner of its data, meaning that all updates on data belonging to a site must be delegated to that site. In more extreme situations, a master site can be defined that owns all the data updates of all the satellite sites, ensuring consistency and read availability over write availability.
  - **Partition tolerance:** All sites use asynchronous replication to maintain local copies of the entire data set. In the event of network partition between the sites, each site can continue to read/write its own data even when other sites are not available. It can also read other sites' data but is unable to change any data that is owned by other sites. In that context we give away some level of partition tolerance for consistency.
  - **Availability:** Local failures are dealt with through the same model described above. Updating data belonging to other sites is blocked. In this context, some degree of availability is sacrificed for consistency.

  Notes: 1. Another option for dealing with consistency under concurrent updates between two sites can be based on versioning, where it is assumed that the latest version wins, or the system delegates the decision on conflicting updates to the application. This is a more complex scenario that goes beyond the scope of this paper.

  2. For more detailed information on how the GigaSpaces topology handles extreme failure, see Nati Shalom's Blog: NoCAP Part III: GigaSpaces Clustering Explained and Elasticity for the Enterprise -- Ensuring Continuous High Availability in a Disaster Failure Scenario

Conclusion

One of the core principles of the CAP theorem is that you must choose two out of the three CAP properties. It is often argued that the right set of tradeoffs for building large scale systems is to give up Consistency for Availability and Partition tolerance. However, in many transactional systems, giving up consistency is either impossible or yields massive complexity in the system design. This paper suggests a different set of tradeoffs in order to achieve scalability without compromising on consistency. It also argues that rather than choosing only two out of the three CAP properties it is possible to achieve various degrees of all three. The degrees are determined by the most likely availability and partition tolerance scenarios of a specific application. The model presented is based on GigaSpaces architecture, derived from years of experience and successful deployments in mission-critical systems for enterprises in the financial services, telecom, and e-commerce industries.

It is important to note that this paper deals mostly with the C in CAP, and not CAP in its broadest definition. There is no intention to dissuade use of solutions based on CAP/eventual consistency models, but rather to caution again adopting such solutions without first considering the implications and alternative approaches. There are potentially simpler approaches to write scalability, such as using database shards, or in-memory data grids.

It is my hope that through this type of sharing of experiences we will come up with a broader set of patterns for building large scale systems that also apply to mission-critical transactional systems.

References

- Lessons from Pat Helland: Life Beyond Distributed Transactions
- The True Meaning of Linear Scalability
- Consistency Models in Nonrelational DBs
- Foursquare's MongoDB Outage
- Why Existing Databases (RAC) Are So Breakable!
- Problems With CAP, and Yahoo's Little-Known NoSQL System
- CAP Confusion: Problems With 'Partition Tolerance'
- You Can't Sacrifice Partition Tolerance
- Gilbert And Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. (2002)
- DeCandia et al. Dynamo: Amazon's Highly Available Key-Value Store. (2007)
- Errors in Database Systems, Eventual Consistency, and the CAP Theorem (Dr. Michael Stonebraker)
- Will Scalable Data Stores Make NoSQL a Non-Starter?

About GigaSpaces

GigaSpaces Technologies provides a new generation of application virtualization platforms. Our flagship product, eXtreme Application Platform (XAP), delivers end-to-end scalability across the entire stack, from the data all the way to the application. XAP is the only product that provides a complete in-memory solution on a single platform, enabling high-speed processing of extreme transactional loads. XAP was designed from the ground up to support any cloud environment – private, public, or hybrid – and offers a pain-free, evolutionary path from today's data center to the technologies of tomorrow.

Hundreds of organizations worldwide are leveraging XAP to enhance IT efficiency and performance. Among our customers are Fortune Global 500 companies, including top financial services enterprises, telecom carriers, online gaming providers, and e-commerce companies.

U.S. Headquarters
GigaSpaces Technologies Inc.
317 Madison Ave, Suite 823
New York, NY 10017
Tel: 646-421-2830
Fax: 646-421-2859

U.S. West Coast Office
GigaSpaces Technologies Inc.
101 Metro Drive, Suite 350
San Jose, CA 95110
Tel: 408-878-6982
Fax: 408-878-6149

International Office
GigaSpaces Technologies Ltd.
4 Maskit St., P.O. Box 4063
Herzliya 46140, Israel
Tel: +972-9-952-6751
Fax: +972-9-956-4410

Europe Office
GigaSpaces Technologies Ltd.
2 Sheraton St.
London, W1F 8BH, United Kingdom
Tel: +44-207-117-0213
Fax: +44-870-383-5135