

Data Consistency and SOA: Old Challenges Rear Their Ugly Heads

Daniel Sholler, W Schulte

Service-oriented architecture (SOA) has modified the designs of many applications. SOA systems are inherently much more distributed than their traditionally designed counterparts. The effects of this distribution are subtle, but are most noticeable when looking at the nature of the data that is used by these systems. This change in design challenges the longstanding assumptions equating data consistency with process integrity.

Key Findings

- SOA designs inherently have distributed data.
- The scope and scale of modern SOA make traditional methods of data consistency management expensive.
- New designs can no longer rely on old assumptions about the relationship between data consistency and the integrity of the process outcome.

Recommendations

- Build explicit consistency assumptions into application requirements.
- Identify information boundaries, particularly where replication occurs.
- Use information flow models to map consistency requirements.
- Build defensive compensation techniques into services.

TABLE OF CONTENTS

Analysis	3
1.0 Introduction: Sharing Data	3
2.0 Why Shared Data Becomes More Difficult With SOA	3
2.1 Data Consistency and Business Outcomes	4
2.2 What We Know About Consistent Data and Consistent Outcomes	5
3.0 Why Data Consistency Is a Problem for Distributed Systems	5
3.1 CAP Theorem	5
3.2 Definitions of Consistency Aren't Consistent	6
3.3 Boundaries	7
3.4 Client Versus Server	8
4.0 Data Consistency Models	8
4.1 ACID	8
4.2 Eventual Consistency	8
5.0 Likelihood of Error and Cost of Repair	9
5.1 Questions of ACID Versus Eventual Consistency	9
6.0 General Recommendations for Data Consistency Uses	9
6.1 Use ACID Transactions When There's a Reference Copy of the Data	9
6.2 Use Mediated Eventual Consistency to Manage Distributed, Low-Volatility Data, Such as Master Data	10
6.3 Build Explicit Consistency Into the Processes When Managing Complex, Long-Running Interactions	10
6.4 Build Services to Tolerate Repetition	10
7.0 Recommendations	11
7.1 Build Explicit Consistency Assumptions Into Application Requirements	11
7.2 Identify Information Boundaries, Particularly Where Replication Occurs	11
7.3 Use Information Flow Models to Map Consistency Requirements	11
7.4 Build Defensive Compensation Techniques Into Services	11
Recommended Reading	11

1.0 Introduction: Sharing Data

Our modern technological world is changing several of the long-held assumptions that we have had about how to use technology. IT practitioners who fail to adapt to these changes run the risk of being left behind, or making serious errors. Many of these changes are quite subtle, and require one to rethink several fundamentals of system design. Among the most far-reaching of these changes is the impact of SOA, cloud, and distributed systems on our notion of data consistency.

2.0 Why Shared Data Becomes More Difficult With SOA

Most corporate computing requirements are based on the notion of recording some information of interest, then communicating that information with different people and with other systems. Underlying this basic concept is a set of inherent assumptions about the information. One of those assumptions is that the information is correct, meaning current, up-to-date, the latest that is known. However, it is not always easy to ensure the correctness of information and its currency. In many cases, systems will generate erroneous results or actions because they display information that is inconsistent. The consistency of the business outcomes in most cases relies on the consistency of the data. Data consistency has been a critical area of research for computer scientists since the start of the computing era. However, for the past several decades, most application programmers have been able to manage with relatively simple assumptions about data consistency. These assumptions (that data exists only in one place, that one can use database transactions as a method of ensuring integrity of business actions, etc.) have been a tremendous convenience in terms of simplifying the code needed to manipulate data.

Early on, data was the "property" of just one application. Clearly, in this case, the application controlled all elements of the data and had whatever consistency model was appropriate for its function. The rise of databases created a system that supported a uniform consistency model — atomicity, consistency, isolation, durability (ACID) property transactions — and it was the responsibility of the application programmer only to define the transaction boundaries in a manner that was consistent with the behavior of the application. In this case, the designers and developers of applications did not have to wrestle with many of the complexities of how to manage data; they could treat the data store as a single point and the transactions as atomic. This central point also coincided with very simple governance models for the data, and the controls based on those governance rules were easier to enforce because of the single source of the information.

Although a great convenience, this centrality of the database was a source of problems, as specific applications owned the particular representation of the data, and other applications were unable to use it in those forms. This gave rise to a whole series of replicas of the information in the form of other databases appended to other applications. This has created several extremely complex logical data structures where the total information know about a subject, for example, customers, was spread across multiple databases, stored according to all sorts of different schemas, and validated according to different rules, etc. This complexity also made it very difficult to modify the data structures, or their content, because the dependencies and implications of any changes were unknown and difficult to track down.

While this duplication enabled the decoupling of those application life cycles, it also made it nearly impossible to maintain consistency among the different stores of the data. There were inconsistency issues among the various applications, but since applications were targeted at

specific user constituencies, this did not have a large practical effect, except when analysis was performed across the entire body of information. This gave rise to data-warehousing techniques to reconcile the inconsistencies that were created by the partitioning of the information.

Improvements in data warehousing and other data management techniques have enabled these processes to become near-real-time, and have spawned concepts such as operational data stores and enterprise information integration (EII). However, in recent years, the focus of information processing has shifted somewhat, and today many organizations are looking to build a library of shared capabilities (shared services) using SOA techniques, and to incorporate them into applications and processes that have been orchestrated across many of the existing system, data and organizational boundaries. This demands an abstract information model that spans the entire organization and the means of representing those boundaries in an application-neutral fashion. Also, this combination of SOA, business process management (BPM) and end-to-end automation has caused many organizations to face large-scale distributed-system challenges, even within their internal enterprise system portfolio. These distributed systems, which often have many redundant copies of information stored in different subsystems, do not follow the comfortable assumptions that we had in the past. Application programmers become responsible for managing many aspects of data consistency, and must do so in a highly distributed environment where that activity is particularly challenging.

2.1 Data Consistency and Business Outcomes

One of the major difficulties in even thinking about this issue is to overcome the association between the consistency of data and the consistency of outcomes. The goal of any process automation system is to have every instance of a process run to completion in a known state. When this fails to happen, there is a serious program error or exception. While this sounds quite obvious, it is very challenging in practice, since in a complex, asynchronous interaction that is part of a long running business process, the number of potential states can be vast. Most process orchestration engines, particularly those that are associated with integration systems, are used primarily to manage the state of those interactions, and to make sure that the process can exit from any indeterminate state, for example, if the further execution of the process is waiting on input from a system, and that system has failed (crashed and rebooted). The process automation system must have the facilities to enable the system to rejoin the interaction and to allow it to complete.

Functionally, the integrity of the business action was maintained by the data integrity mechanisms. In the simple case of a single database with ACID properties, the transactional mechanism in the database and the record of business interactions are presumed to be one and the same, thus the technological mechanisms, such as commit and rollback, can address the requirements. In other words, one assumes that the business action generates a singular record that is recorded in the database, and that the boundaries of the database transaction determine the boundaries of the system transaction, and are presumed to be the same as the business transaction. In this case, the state of the data would be very clear (because it would be current to the last observed transaction boundary).

In the distributed case, neither the clarity nor the indicators may work properly. For example, when activating a process that contains, for example, a dozen steps, these steps might be implemented by several different applications or services operating independently. While each is aware of its state, the composite state is likely to be invisible. Coordinating these actions so that they are apparently atomic is one possibility, but requires substantially more processing, and in some cases substantial exchanges of information to coordinate.

As processes become increasingly distributed in their execution, they are less able to rely on data consistency as a means of preserving the consistency of the outcomes. Organizations are forced

into making some implementation and design decisions based on the type of consistency that is desired and the method of enforcing that consistency. These kinds of design implications are unfamiliar to many developers, and will demand careful attention as the systems are built.

2.2 What We Know About Consistent Data and Consistent Outcomes

No matter what the system or design, if the system is distributed, the designer is always faced with choices about consistency. The first choice is basic: a single point of information (and, therefore, a single point of failure) versus distributed data. In reality, in most modern SOA environments, the data is, by definition, distributed. This is usually true because applications insist on having a data store that they can treat as private for their purposes. If these applications are purchased, there is effectively no way to have them operate in the absence of their own copy of the data. Once the data has been copied, there is always a question of how to maintain consistency among the copies and of what the business consequences of that inconsistency might be.

The issue that faces application designers is how to manage the "window of inconsistency" so that it has few or no adverse business outcomes. This decision is highly dependent on an evaluation of the process and the behaviors of the individuals within that process. For example, although it might be reasonable for a bank website not to reflect deposited checks until the following day, most users would expect it to register electronic transactions as they are entered. From the point of view of the Web application, these two things might be identical (in other words, they are just transaction records that the application sources from the current account system), but they have qualitatively different expectations of currency in the users' minds.

There are also situations in which relaxing the definitions and requirements of data consistency can either radically change the cost of implementing a process, or can change the process itself. Interesting cases of this are travel resellers such as Priceline.com. Travel industry processes have always assumed that the prices and dates of travel need to be fixed prior to the execution of a sales transaction. Priceline.com's business model is based on a radical change in the business process where the price is negotiated and the date of travel is committed.

Finally, there is a question of what elements of the interaction must be recorded. For example, when a sales transaction is initiated, one of the first things that usually happens is the negotiation of a price. In most sales systems, details of the state of the negotiation are rarely recorded. This is a choice made by the designer of how to correlate the state of the IT system with the real-world state. Historically, we have allowed systems to ignore the state of these kinds of activities because they were considered "outside" the realm of the system's function. However, as we use SOA to build more-extensive end-to-end automation, more and more of those usually undocumented states are encompassed by the system. Furthermore, the linking of separate systems creates increasing discrepancies between the state of the data and the state of the real-world business interactions.

3.0 Why Data Consistency Is a Problem for Distributed Systems

3.1 CAP Theorem

Most technologists who have faced challenges with data consistency have observed that there is often a trade-off to be made between the consistency of the data and its availability. The first person to formalize this trade-off was Eric Brewer (see www.cs.berkeley.edu/~brewer) in a discussion of what he referred to as the consistency, availability and partitioning (CAP) theorem (see www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf). This theorem has been validated by others (see <http://portal.acm.org/citation.cfm?doid=564585.564601>). His theorem is that one can only have two of those characteristics in a performant system. In other words, as we

move toward SOA (with its implicit use of partitioned data), we are faced with a trade-off between data consistency and data availability. This is a difficult decision, because from one point of view, the system should always operate using the latest (i.e., most correct) version of the data. This is the property that ACID enables. No matter what is done, the data seen is the latest version, and it is assured that there will not be changes to the data while it is being read or written. However, in many situations, it is clear that having some capability available to the user is more important than having consistent information. For example, an airline reservation system may have booked a particular seat on a particular flight. However, because there are many copies of that data, an alternative system (for example, the Web-based system) may allow someone else to book that same seat, because its copy of the data is not consistent with the other. From an IT perspective, this is undesired, but from a business perspective, it might be quite reasonable, since the likelihood that this will cause a real-world problem is remote (in most cases, there will be a cancellation that will allow the person to get a seat, if not the one specifically requested).

The idea that we can relax ACID property constraints on our data is a powerful one. On the one hand, it means that complex shared data scenarios, including shared services, and complex business process orchestrations can be achieved without making radical changes to the design of the data-owning systems. On the other hand, this transfers responsibility for the integrity of the business outcome from the infrastructure to the process of interaction. In other words, it makes integrity visible at the application level. With the ACID assumption, the integrity of the outcome could often be assumed due to the inherent integrity of the data. In these cases, there are many more-intermediate states.

3.2 Definitions of Consistency Aren't Consistent

The issue is that consistency is not an absolute property, but only for the given purpose. For example, a service that shows flight information might show the schedule as planned weeks in advance, or it might show the planned work schedule for that day (which starts out the same as the original plan, but changes based on weather delays and aircraft availability), or it might show up to the second flight status (which may vary based on even more fine-grained factors, such as the timeliness of the food services crew). In each of these situations, the information syntax is the same, but the meaning of the scheduled time is slightly different, due to differing assumptions about the consistency of the data.

Obviously, in some cases, these differing assumptions will not matter. For example, if a person is trying to buy a ticket for tomorrow, a difference of 10 minutes in the scheduled departure time is not that relevant, whereas if a person is trying to make it from the counter to the gate, 10 minutes matter quite a lot. In the first case, if the ticket agent cannot see the 10-minute delay caused by the late food service delivery, there is no consequence to the business process. However, in the second case, the person will learn that the flight has already left, when there may still be a chance to make it, which is (potentially) a significant consequence. The reason that these differences in interpretation sometimes exist is because information structures created for one purpose are used in other purposes. Because all of this data looks the same (i.e., flights and departure times) there is an assumption that it can or should all be handled the same way. However, it is clear from these simple use cases that the underlying assumptions sometimes matter. This leads to a situation where we must know about all the information currency assumptions.

The second problem is that consistency in a distributed environment can take many forms. One common form of consistency ("monotonic read consistency") says that a person will never read anything that is older than the last thing read. For example, if data and query loads were distributed across multiple copies of a database, and updates were applied to those copies in an unpredictable order at unpredictable times, it would be possible for an application to first read from an updated copy, then repeat the same read, but be routed to a copy that has not yet been

updated, yielding an "older" result. From the point of view of the application, the data has changed; in reality, what the application sees as a forward change (first the person saw A then B, so B must be the most current) is exactly the opposite of what occurred. Systems that enforce monotonic read consistency will not allow the application to see result B, and will instead return the later time stamped as result A.

While this system does not assure that viewers will always see the latest data (it could have been possible for the system to see B first, then A), it does assure that when viewers see a change, the changed value is not an older artifact, but a change that has taken place while viewers were examining the data, which can be an important assumption in many cases.

There are many more forms of consistency, which are summarized nicely in a blog entry by Amazon CTO Werner Vogel (see www.allthingsdistributed.com/2008/12/eventually_consistent.html#more). The reason it is important to understand these is that although database designers spend a lot of time on this issue, it is now becoming important for SOA application developers to build and apply consistency mechanisms within their software. This goes back to the point made in the previous section: If the infrastructure is not guaranteeing data consistency, then issues of state need to be handled at the application layer. This is not all that unusual; in most cases, the application layer needs to be aware of potential failures when information is transferred from one physical component to another. The difference is that SOA designs dramatically increase the number of these component interactions, so instead of programming each one "from scratch" as a special case, it is useful to think of an overall design that accommodates them.

In addition to the basic consistency model, other software and data services may affect the ability to maintain consistency. Many of these have to do with timing. A simple service that accesses some data may actually invoke many stored procedures in the database, require the materialization of views, etc. All of these actions put a boundary on how fast some activity may occur. If that window is not fast enough for one of the use cases, then that use case must introduce the idea of operating with inconsistent data.

3.3 Boundaries

Another challenge when partitioning data has to do with the differences in the different databases. In most cases, the data has been replicated because some boundary has been crossed. This may be an organizational boundary, where different parts of the organization have different use cases for information, or it may be some other boundary that causes the design to be reproduced, such as having hired a different system integrator (SI) partner to assist with the development of two different applications which share some information. Usually, this boundary included some technology requirements (for example, the items in the replica were designed to work with a particular application that used Microsoft technologies, and ADO .NET, for example). However, there are also issues of information stewardship and trust. In the single-source model, the owner of the data was presumed to have governance responsibilities. In an SOA environment, where the information associated with an element of the corporate information model may be spread around the organization, it is critical to be more explicit in terms of governance roles and quality requirements.

While we may logically accept that all the customer information should be the same, allowing someone else the physical management of critical customer data (even if that someone else is within your company) is a difficult decision. Because many of the replicas that exist do so because of these trust issues, one of the items that is usually a prerequisite for dealing with any distributed data environment is identity management, so that the actions that are taken on information can be tracked across the copies of the information and across organizational boundaries.

3.4 Client Versus Server

Another difficult issue is figuring out who is responsible for determining the correct state. In the classic ACID database model, this is clearly the server. The database serializes the requests from the various applications and users, and manages the state. However, many other models can be utilized. Most Web applications use a slightly different assumption (derived from the representational state transfer [REST] principles) where the information that is owned by the client is always assumed to be current, even if it was transferred from an outdated copy on the server.

Many systems attempt to solve this problem by handing the complete state back and forth among the actors. However, this is impossible when the actors are independent programs that have different representations for the state, such as when integrating two unrelated packaged applications. This avoids the problems associated with hidden state, but it still does not identify how to resolve change conflicts, where multiple actors change the state in different ways at the same time. In order for this to occur, there must be some kind of data consistency model that is understood to apply in the circumstances. The processes that make use of this data must be aware of the choice of model, and must develop their actions based on the assumptions in that model.

4.0 Data Consistency Models

Since the actual goal is to ensure the consistency of a business outcome, there are many different models that allow that to occur. The ACID model that most modern databases support is just one possible model, and even within that there are variations (the difference between "repeatable read" and "cursor stability" modes in SQL, for example).

4.1 ACID

Most developers are familiar with the notion of ACID, which is an acronym standing for:

- Atomicity (everything either succeeds or fails together; there is no partial work)
- Consistency (data is in a consistent state both before and after the transaction, regardless of success or failure)
- Isolation (no one can see the data while it is inconsistent; i.e., being written)
- Durability (when the transaction over the state is permanent, and no type of failure can undo it)

These are the qualities that most developers associate with the notion of a transaction. However, data consistency is just a means to an end. The range of things that may qualify as "consistent" and the consequences of inconsistencies are highly dependent on the cost of maintaining the consistency, and the value that consistent data has to the organization.

4.2 Eventual Consistency

In many cases, it is both cost-effective and reasonable to have what has been termed "eventual" consistency. This can take several forms, but the notion is that, as the term describes, the system will eventually become consistent. The idea with eventual consistency is that, in a nonfailure state, the window of inconsistency (the time during which reads from copies of the data might return different results, because one of them is not yet updated with the new values) is relatively predictable. This means that within certain parameters, processes can rely on the window duration. In many systems, there are eventual consistency models, the most common of which is

the data consistency form of integration (see "Understanding the Three Patterns of Application Integration"), which is much easier (and less-expensive) to implement, and which may serve the process well.

5.0 Likelihood of Error and Cost of Repair

Obviously, one of the determinations that affects facets like the inconsistency window, as well as whether the system supports eventual consistency, has to do with the overlap between the read and write sets. In a situation where only a small percentage of the database is read and written at any time, and where the read will be evenly distributed throughout the data set, the probability of interference can be reasonably calculated. While some processes tolerate no errors, most processes can deal with a situation where the wrong information is used one time out of tens of thousands. The real issue is: When does the cost of the repair associated with this error rate become significant enough to overshadow the cost of implementing some kind of ACID data consistency? In the past, when faced with this problem, most organizations would focus on consolidating the data into a spot where they could use the database engine. However, SOA applications often make this impossible, because they utilize shared components that have their own inherent assumptions about where the data is and what it means, so the focus is on either reducing the error rate or reducing the cost of repair.

5.1 Questions of ACID Versus Eventual Consistency

There are many who believe that in a complex, highly distributed SOA environment, the notion of strong (ACID) consistency is just not useful. This thinking is based on the assumption that in any situation where consistency of information could impact the business outcome there will be code to handle failures, because it is always possible to fail to acquire the correct or complete information for reasons that are nontechnical. For example, it is possible that people's names in a database were misspelled or even willfully misrepresented. In fact, in many cases, these types of errors are far more common than the errors that developers worry about, such as failures of communication or computers, or of being locked out of data waiting for a consistency update. Given that there must be some code to deal with the consequences of the logical errors, the marginal cost of adding code to deal with the IT system failures (including the failure to use the latest data) is very small. In some designs, this can even be zero. This implies that spending lots of money and effort to ensure data consistency at the underlying layers (for example, the database)

6.0 General Recommendations for Data Consistency Uses

Ultimately, organizations must make some decisions about how and when to enable data consistency across distributed data. In this section we list some general rules.

6.1 Use ACID Transactions When There's a Reference Copy of the Data

In the SOA environment, the partitioning of the data is nearly always present. However, there are often cases where, from a business point of view, even though copies of the data may be (at any given time) out of sync, there is a business agreement that one privileged copy is used as an authoritative source. The use of an authoritative source implies that the data is partially consolidated around that source, and usually means that the mechanisms for ensuring integrity (such as a two-phase commit) have already be implemented and tested with that data source. In these cases, the services that enable the manipulation and use of that data should be able to hide virtually all consistency issues from the use and utilize ACID data consistency to manage the integrity of the outcome.

6.2 Use Mediated Eventual Consistency to Manage Distributed, Low-Volatility Data, Such as Master Data

One very common scenario with SOA is the creation of a set of services for managing various classes of master data. Whether conceived as part of an SOA project or as part of a master data management program, this model is an excellent situation in which eventual consistency plays a large role. Nearly every master data management scenario has portions of the total master data replicated near the applications that consume that information and a central store (either materialized or virtual) that coordinates the changes between them. Changes are communicated among the participants asynchronously, and that enables them to remain performant and rapid. However, the cost is that there is some small window of inconsistency. Given the low volatility of master data, the likelihood that someone will read the data in that window is usually very small. Therefore, this model enables a highly scalable means of improving data quality and consistency, without putting all the applications using the data into a synchronization straightjacket.

6.3 Build Explicit Consistency Into the Processes When Managing Complex, Long-Running Interactions

In cases where SOA services are used as elements in a BPM suite (BPMS)-based business process, this is usually a circumstance in which eventual consistency is required, and where the consistency management needs to be explicitly included in the process definition. The relationship between the logical model of the business transactions and the consistency of the data needs to be clearly defined. These processes generally run over lengthy periods (minutes, hours, days, weeks). The coupling among the various actors is loose and is intentionally so. The process itself must account for a variety of logical errors, which can easily add consistency errors. The only meaningful technique involves building a set of compensations for each action undertaken by the process. If each action has a compensating action (or if actions can be grouped into a set, and the set have a compensating action), this will allow the process to be unwound at any point.

Compensations are not always simple. In an ACID system, one can rely on the "rollback" action, since no one else has access to the data. However, in more-complex systems, especially in distributed systems with multiple asynchronous parts, the compensation may be entirely different from the action itself. For example, the compensation for sending an invoice to a counterpart is likely to be a complex series of steps that determines whether the invoice was sent (and, therefore, a change process initiated with the counterpart), or whether it can just be deleted from the local sending queue.

Different tools have different approaches to how compensations are represented, or more accurately, how the relationship between an action and its compensation is represented, and how the compensation is triggered. Systems that do a good job of helping users create compensations and trigger them when appropriate are usually easier to use than those that require the compensation to be an explicit path in the defined process.

6.4 Build Services to Tolerate Repetition

One of the classic "defensive programming techniques" that can be used is to design services so that transactions may be transmitted multiple times without error. Although many techniques allow this, being able to resend an action to another system without adverse consequences makes it quite simple for any actor to recover from any sort of failure. It can simply repeat its actions, and as long as those actions have the same outcomes, no harm will come.

7.0 Recommendations

7.1 Build Explicit Consistency Assumptions Into Application Requirements

A proper process definition will include the requirements for consistency, the assumptions about inconsistency windows and the compensations required.

7.2 Identify Information Boundaries, Particularly Where Replication Occurs

Where information can exist in more than one place, that fact should be known, and any systems that make use of that information need to understand what kind of consistency model is in place, and how to use the data in a manner that is in line with that model.

7.3 Use Information Flow Models to Map Consistency Requirements

Strong consistency is very expensive in the distributed world, but business failures due to inconsistent data may be expensive as well. The goal is to create a level of consistency that is commensurate with the likelihood of error and the cost of remediation. This can only be modeled with a complete understanding of the information flow.

Determine at what level the consistency should be maintained.

7.4 Build Defensive Compensation Techniques Into Services

If services are programmed to assume that data inconsistencies and resynchronizations will need to occur, the systems will be much more robust. Even if these capabilities are not exercised frequently, they will allow systems to run even in the face of erroneous data.

RECOMMENDED READING

"Q&A on SOA and Latency in Applications"

"The Emerging Vision for Data Services: Logical and Semantic Management"

REGIONAL HEADQUARTERS

Corporate Headquarters

56 Top Gallant Road
Stamford, CT 06902-7700
U.S.A.
+1 203 964 0096

European Headquarters

Tamesis
The Glanty
Egham
Surrey, TW20 9AW
UNITED KINGDOM
+44 1784 431611

Asia/Pacific Headquarters

Gartner Australasia Pty. Ltd.
Level 9, 141 Walker Street
North Sydney
New South Wales 2060
AUSTRALIA
+61 2 9459 4600

Japan Headquarters

Gartner Japan Ltd.
Aobadai Hills, 6F
7-7, Aobadai, 4-chome
Meguro-ku, Tokyo 153-0042
JAPAN
+81 3 3481 3670

Latin America Headquarters

Gartner do Brazil
Av. das Nações Unidas, 12551
9º andar—World Trade Center
04578-903—São Paulo SP
BRAZIL
+55 11 3443 1509