Global Technology Services

Best Practices for Deploying a CMDB in large-scale Environments

Alexander Keller, *alexk@us.ibm.com*
Suraj Subramanian, *suraj@us.ibm.com*

IM 2009: 11th IFIP/IEEE International Symposium on Integrated Network Management

Abstract

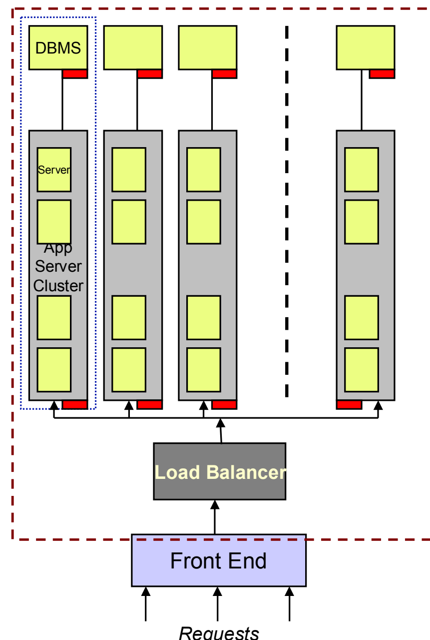
We describe best practices for deploying a Configuration Management Database (CMDB) that we have developed during several recent client engagements. Given the complexity and novelty of CMDB solutions that deal with discovering, storing and tracking actual Configuration Items (CIs), many enterprises rely on service delivery organizations - such as IBM Global Technology Services - to perform the configuration and roll-out of the system into production. This can be either done on the customer premises (within the scope of a so-called project-based service engagement), or by subscribing to a managed service, and thus leveraging the IT service management environment that the service provider has already set up.

Often, enterprises severely underestimate the effort involved in setting up IT service management infrastructures by mistakenly equating the setup of such a complex system with the mere installation project of a shrink-wrapped, self-contained product. This, however is not the case: The immense heterogeneity of data center resources makes that no single vendor can cover the breadth of managed resource types when new product versions ship every 12 months, often by means of integrating acquisitions into the product portfolio. Consequently, today's IT Service Management systems rather resemble construction kits and frameworks that require a good deal of tailoring and customization to become useable and useful to the customer. The present paper attempts to provide an insider view into the issues that a CMDB deployment architecture needs to address. In our work, we found that the success of a CMDB deployment project can be attributed to a set of tradeoffs and best practices, especially when it comes to tuning the performance of the system and orchestrating the distributed components of a CMDB so that they work well together.

By grounding our work in a concrete case study and by referring to real-life requirements, we demonstrate how to develop an operational architecture by using an off-the-shelf CMDB product. We point out the key design points of our architecture and describe the tradeoffs we had to make, which we subsequently distill into a set of best practices that have been successfully applied in sizing, estimating and implementing subsequent CMDB deployment engagements.

Global Technology Services IBM

Motivation: Tracking Dynamic Data Center Topologies



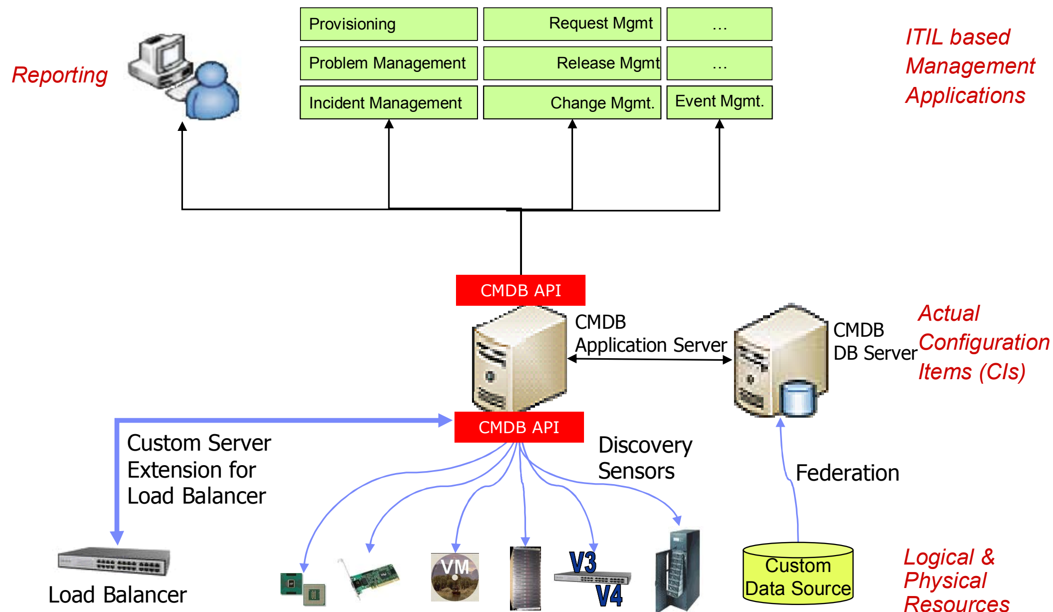
- 10,000 – 20,000 servers:
  - Windows Server 2000 and 2003
  - Solaris/Sparc (large DB servers)
  - Solaris/i86
  - Linux (often on Blade Servers)
- 3-6 data center locations
- heavy use of load balancers: ~800 – 1400
- Every application tier is front-ended by a load balancer
- Every server in the environment belongs to a load-balanced pool
- Load balancer pools get reshuffled every few hours
- Load balancers are deployed pairwise:
  - Primary
  - Backup

Slide 2 Best Practices for Deploying a CMDB | IM 2009: 11th IFIP/IEEE International Symposium on Integrated Network Management © 2009 IBM Corporation

One of the major challenges in today's data centers consists in discovering, tracking and auditing the configuration and relationships of several thousand heterogeneous servers and the operating systems, middleware and software they host. Configuration Management Data Bases (CMDB) are widely regarded as a means to achieve this goal. Deploying CMDBs in large-scale environments, however, brings its own challenges: Due to the need for adapting the data center infrastructure to continuously changing workloads, topologies are becoming increasingly dynamic, which requires frequent refresh cycles of the information contained in a CMDB. On the other hand, a CMDB needs to keep track of the Configuration Items (CIs) it stores at a certain level of detail in order to be useful. It is the tradeoff between the level of detail and the freshness of information that makes CMDB implementations particularly challenging in practice.

The above figure depicts an increasingly common data center topology: while the separation of a distributed application in a variety of tiers (front-end, application, database) has been common practice for quite some time, we increasingly see that every tier is load balanced, in order to swap servers in and out of a load balanced pool. Load balancers play a special role because they provide a means for efficient workload-to-resource allocation by directing traffic to systems that have spare capacity available. From a management perspective, load balancers are a valuable source of management information because they keep track of the utilization and health of the servers that are part of the load-balanced pools. As such, load balancers are excellent monitoring points from which a resource discovery application (or a CMDB) can learn about the existence and status of servers without going directly to the servers.

This paper is structured as follows: we first introduce a typical customer scenario on slide 2 and identify the functional and non-functional requirements on a CMDB on slides 3 and 4, respectively. On slide 5, we describe the distributed architecture of Tivoli Application Dependency Discovery Manager (TADDM), a commercial CMDB implementation with a focus on configuration item discovery and tracking. Subsequently, we focus on the design and performance engineering of our distributed solution (slide 7) and list the best practices we have identified (slides 8 – 11). Finally, an evaluation of performance is presented on slide 12. We, conclude the paper by summarizing key findings and by pointing out areas for further research.



A lot of effort is often spent in integrating the CMDB with already existing infrastructure and applications. The central role that the CMDB plays for an enterprise is illustrated by the following, typical functions it performs:

- **Discovery Capabilities**: a CMDB typically ships out-of-the-box with a set of agent-less **discovery sensors** that allow it to explore the details of a managed resource type (e.g., pools of a load balancer, or the volumes of a storage array). In addition, a CMDB may be able to connect to a set of SNMP-enabled or proprietary agents that are located on the managed systems in order to gain access to relevant data. However, due to the large amount of heterogeneous resources, vendors often ship software development kits that allow the customer to create **custom server extensions** for managed resources that are not covered by the standard set of sensors. Configuration Items that are discovered as often referred to as **actual CIs**.
- **Persistent storage in RDBMS**: while today's CMDBs follow an object-oriented approach – sometimes derived from the Common Information Model (CIM) – the data typically gets persisted in a common-off-the-shelf relational database management system (RDBMS). While this typically results in running the RDBMS on a separate server than the CMDB application, an additional implication is that an object/relational mapping needs to be performed. This, in turn, may impose limitations on integrating existing, custom RDBMS-based data collection tools in the CMDB, e.g., by means of federation.
- **Reconciliation**: It is well possible that a managed system is discovered by more than one sensor (e.g., a server gets discovered through a native operating system-level sensor and through an SNMP sensor). Whenever this is the case, the CMDB needs to reconcile the different views into a single resource. To do so, it follows a set of rules that govern the identity of a managed resource: e.g., a combination of serial number, MAC address, IP address for a computer system.
- **Baselining and Reporting**: at any given point in time, a snapshot of the CIs can be taken to form a configuration baseline against which audits can be performed. A suitable way to carry out audits is by means of generating custom reports, which often create significant workload on the CMDB.
- **Programmatic Access to CIs via CMDB API**: the fact that the CMDB stores authoritative configuration data in a normalized format makes it a prime data source for a variety of management applications that implement the ITIL [1] best practices (depicted at the top of the figure). Driving tools off the CMDB adds further workload.
- **Dependency Visualization and Topology Maps**: the graphical representation of logical and physical topologies and the browsing of dependency relationships are often viewed as key features of a CMDB. In practice, where we regularly encounter CMDBs that have several millions of CIs, we found that great care has to be taken to restrict the amount of CIs that will be displayed. Otherwise, the CMDB spends a significant amount of time trying to render a topology that will be of no use to the customer as several thousand CIs and relationships clobber the display canvas.

Global Technology Services

IBM

Non-functional Requirements on a CMDB

- Different SLAs for discovery cycles for different managed resource types:
  - a 6-hour duty cycle for ~400-700 load balancer pairs
  - a 24-hour duty cycle for all the remaining IT infrastructure assets (both Hardware and Software)
- Tools and applications use CMDB as authoritative data source → additional workload on CMDB
- High Availability and Disaster Recovery:
  - Determine at which parts of the system need to address this
  - Devise data replication strategy for redundant systems

  →extra workload on CMDB
- However:
  - a CMDB is not a monitoring tool!
→ it is not intended to cover 'real-time' resource status
  - None of the above requirements are met by today's products out-of-the-box
→ significant amount of architecture and design needed

Slide 4

Best Practices for Deploying a CMDB |
IM 2009: 11th IFIP/IEEE International Symposium on Integrated Network Management

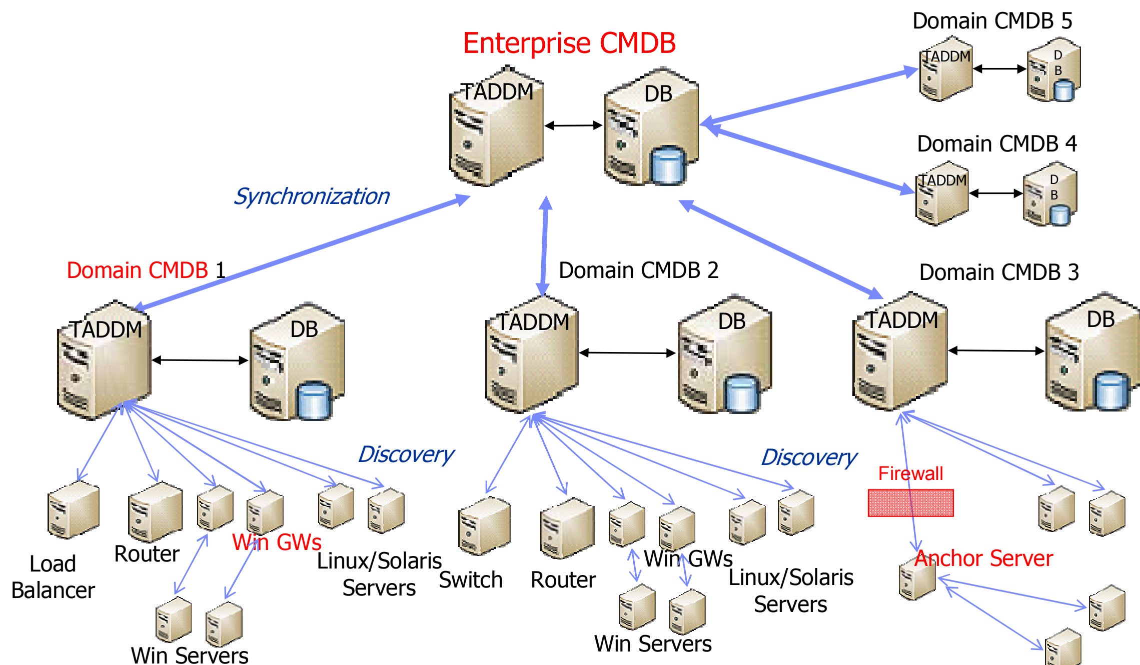
© 2009 IBM Corporation

In addition to being seamlessly integrated into the customer environment, a CMDB deployment needs to address a variety of non-functional requirements:

First and foremost, one needs to determine what timespan between the discovery of a CI and it being made available to the management applications or the graphical console is deemed acceptable. While it is understood that a CMDB will not be able to cover resource status in 'real-time', there is broad consensus that data older than 3 days is often considered worthless, especially if the CMDB is the basis of tools, such as resource provisioning systems. An interesting variation of this problem occurs when different types of resources need to be prioritized, e.g., when load balancer pools get reshuffled every 6 hours, and the CMDB should reflect this while the CIs of all remaining infrastructure assets may be up to one day old. The role of the architecture becomes especially important when such a feature has to be designed around a CMDB implementation that does not support pre-emptive scheduling by suspending/resuming jobs.

Second, the fact that customers drive tools and ITIL processes off the central CMDB makes that the system needs to be sized so that it has enough capacity even when it refreshes its data (or specify a blackout period during which no tools or reports may be run against the CMDB). Reporting in particular places significant workload on the system, which may in some cases necessitate the separation of the CMDB in a 'read-only' (for reporting and tool access) and a 'write-only' (data refresh from the environment) part.

Third, there may be a need to design the system with a focus towards high availability (within a data center) and disaster recovery (across different data centers) if the CMDB is considered mission-critical. Here, one needs to strike a balance between the cost for redundant systems, and whether it is acceptable to only harden the system on which the part of the CMDB is stored against which reports/tools are run.



Tivoli Application Dependency Discovery Manager (TADDM) [4] is IBM's implementation of the parts of a CMDB that relate to the agent-less discovery, tracking and auditing of actual configuration items. As depicted in the figure above, TADDM is a hierarchical, distributed system that can scale to very large, geographically distributed environments. It consists of 4 layers:

1. **Enterprise CMDB (eCMDB)**: This is the top-level 'system of record' for an enterprise in terms of discovered data center equipment. Because an eCMDB does not perform discovery tasks, it is populated with data from one or more Domain CMDBs (see below). Such synchronization with a Domain CMDB can be run from the eCMDB either on a schedule or on-demand. The enterprise CMDB has an API (CLI, java and web services interfaces) that is used to query the consolidated data across the enterprise. In addition, the eCMDB can be queried by means of a report generator.
2. **Domain CMDB (dCMDB)**: its primary responsibility consists in discovering a delimited subset (aka **domain**) of the managed resources that exist in the data center. Every dCMDB needs it own database to store the discovered CIs. Discoveries can be started manually or scheduled from the GUI, or invoked through an API (CLI, java and web services) by an external application. Managed Resources are typically grouped into **scopes** (typically by specifying IP ranges and/or individual addresses). It is important to combine the right scope of devices that need to be scanned or discovered via the dCMDB with the proper **profile** (the level of detail of information gathered); otherwise, discoveries can take a very long time. This leads us to **Tradeoff #1: Discovery Breadth vs. Discovery Depth**
3. **Windows Gateways** and **Anchor Servers**: the former translate the secure shell (ssh) commands issued by TADDM into Windows Management Instrumentation (WMI) commands Windows systems understand; the latter tunnel firewalls by means of ssh so that systems located behind a firewall can be discovered.
4. **Managed Resources**: the systems that need to be discovered: servers, middleware, applications, storage, networking equipment, load balancers etc. TADDM provides about 50 out-of-the-box discovery sensors for a variety of managed resources (e.g., Oracle database, WebSphere Application Server, VMWare, Veritas Storage, SNMP devices etc.). Using sensors implies that the credentials for the managed resources need to be stored at the Domain CMDB level (or access granted by means of ssh keys); a collection of discovery sensors that will be applied to a given scope is called a **profile**.

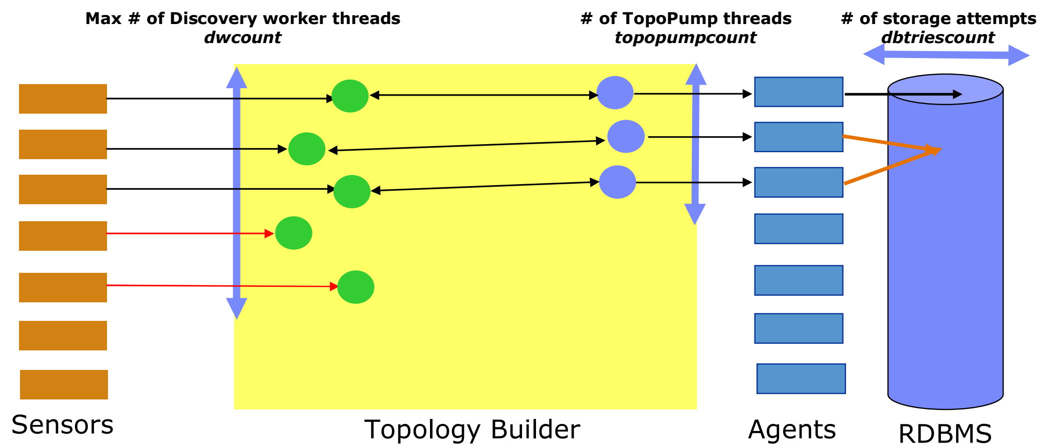
The architectural challenge consists in estimating the right number of Domain CMDBs, Windows Gateways and Anchor Servers so that the solution meets the requirements described on slide 3. In addition, one has to orchestrate the interworking of the various components of TADDM as there are a few limitations that impact the order of tasks:

1. a dCMDB can either run a discovery, or synchronize with the eCMDB, but not both simultaneously.
2. there is no pre-emptive scheduling of discoveries, i.e., there is no graceful suspend/resume of discoveries.
3. there is no concurrent synchronization – the eCMDB can only synchronize with one dCMDB at a time.

From these constraints, we derive **Tradeoff #2: Discovery Time vs. Synchronization Time**

Global Technology Services IBM

Domain CMDB: System Model for Resource Discovery



Performance trade-offs:

- If dwcount and topopumpcount too low → low throughput
- If dwcount >> topopumpcount → Sensors are kept waiting (red arrows)
- If #TopoPump threads large → retries for DB writes and/or storage errors (orange arrows)

Slide 6 | Best Practices for Deploying a CMDB | IM 2009: 11th IFIP/IEEE International Symposium on Integrated Network Management © 2009 IBM Corporation

In order to set the stage for the following performance engineering discussion, it is important to gain an understanding of the Domain TADDM system model by breaking down the overall discovery process into the following two phases:

1. Discovery by means of resource-specific Sensors: The upper bound for the number of concurrently executing discovery worker threads can be set by *dwcount*.
2. Topology building: this phase addresses the reconciliation of configuration items that have been discovered through different sensors, their physical storage in the database, and the creation of dependency topology maps. *topopumpcount* is the configuration variable that sets the maximum number of topology threads; *dbtriescount* is the maximum number of times the topology builder will try to successfully commit a CI to the database. The last step of the topology building process consists in the execution of the view manager to build the topology maps that can be browsed through the GUI.

From the above discussion, it follows that one of the key performance optimization problems around discovery consists in determining the optimum values for *dwcount* and *topopumpcount.* These values, in turn, are heavily influenced by a variety of configuration parameters for the database, Java Virtual Machine, and operating system and – last but not least – by the hardware characteristics of the servers that host the TADDM and database servers.

Tradeoff #3: Discovery Speed vs. Data Consistency

- Increasing *dwcount* makes the discovery go faster because more sensors are executing concurrently.
- Increasing *topopumpcount* increases the rate of CIs stored in the database. However, over a certain level, data consistency is affected because after the threshold set in *dbtriescount* is reached, the system drops the data it cannot store.

While analytic approaches based on statistical and control models have successfully been applied to determine such parameter settings in somewhat related problems, the optimization problem that would need to be solved in the present case is significantly more complex than the one described in [3]: while prior art focuses on explaining the tuning parameters of a single DB2 database server by several counters and gauges, the current case involves not only configuration parameters applying to multiple levels of software within the same system (note that TADDM has discovery sensors for over 50 different resource types, each yielding a different amount of CIs), but is spread across several physical servers. Recall from slide 5 that, additionally, Windows Gateways and/or Anchor Servers may be involved in the discovery process.

In order to meet the requirement of accommodating different discovery duty cycles for load balancers and servers (6h and 24h, respectively) and assuming a size of the distributed environment of about 15,000 servers (the median between the 10,000 to 20,000 servers we frequently encounter in client engagements), we need to design a system that consists of multiple Domain CMDBs that synchronize into an Enterprise CMDB. The topology of the system is similar to the one depicted on slide 5.

We will now recap the deployment tradeoffs identified so far:

1. **Tradeoff #1: Discovery Breadth vs. Discovery Depth:** While we need to identify the details for each system at an OS-level (and thus require at least OS-level discovery – level 2 – capability with credentials), we are able to disable the sensors for which no corresponding resources exist in the environment.
2. **Tradeoff #2: Discovery Time vs. Synchronization Time:** There is an upper bound to the synchronization time that stems from the eCMDB: Given that each Domain CMDB needs to synchronize 4 times per day with the eCMDB, and the absence of concurrent syncs, it follows that *24 hours >> #dCMDB x synctime per dCMDB*. For 6 dCMDBs, this number is 4 hours; for 5 Domain CMDBs, the number is 4 hours and 12 minutes. However, on a domain CMDB level, 4 hours encompass both discovery and synchronization times because a dCMDB can always do only one thing at a time.
3. **Tradeoff #3: Discovery Speed vs. Data Consistency:** on a dCMDB level, we need to make sure that we encounter as few storage errors as possible. The maximum allowable number of *dwcount* is therefore determined by the maximum number of *topopumpcount* that does not produce storage errors.

Comparison of Design Alternatives for Domain CMDB Assignment and CI Reconciliation

Approach 1: Assign different Resource Types to different Domain CMDBs

Using a dedicated domain CMDB for discovering Load Balancers spread across several data centers and one or more dedicated domain CMDBs for servers seems straightforward because it allows to implement different discovery cycles in a seemingly simple way. The major issue that prevents this approach, however, is the fact that the reconciliation between individually discovered servers by means of sensors and the same servers being located in a separately discovered load balancer pool would happen at the enterprise CMDB level. Reconciling 15,000 servers and 1000 load balancers at an eCMDB level will overwhelm the system and make the eCMDB the bottleneck. This lack of scalability alone makes this approach unfeasible in practice; additional drawbacks are listed in the slide above.

Approach 2: Keep Discovery local and upload disjoint data sets to eCMDB

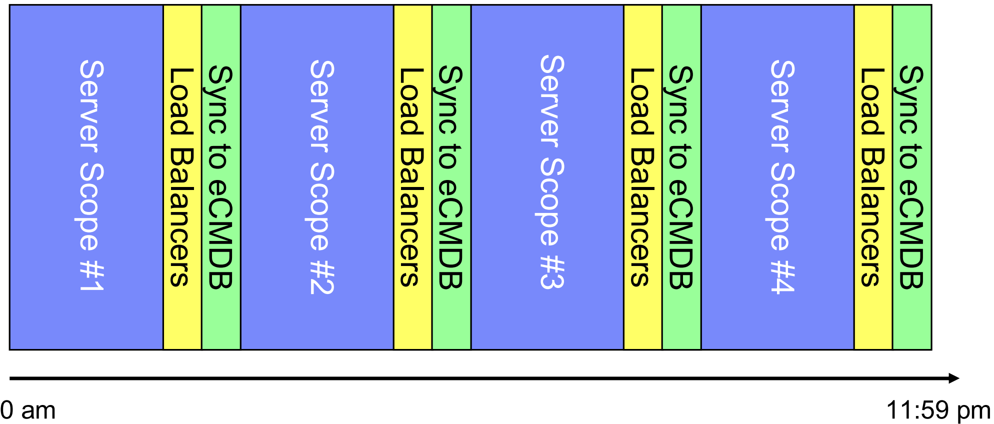
This approach consists in creating schedules for discovering and reconciling load balancers and servers on a per-dCMDB basis. In other words, we define a domain and use a dCMDB for every data center so that all the equipment within the domain (load balancers, servers and other devices) are discovered by the local dCMDB. Data from all the local dCMDB is then consolidated at the enterprise CMDB, and under the assumption that only few inter-data center dependencies exist, the reconciliation effort at the eCMDB level is minimal because disjoint data sets are uploaded to it (i.e., there are no or only very few interdependencies between resources located in different data centers).

While the enforcement of the locality principle keeps reconciliation of managed resources confined to the boundaries of a domain, the additional cost incurred is that one needs to make sure that the servers and load balancers are partitioned into scopes in a way that their discovery will finish within the allocated timeframes. One issue is that there is currently no provision to automatically balance the scope depending on discovery times; instead, it has to be done manually. Consequently, whenever managed resources are newly introduced, they need to be assigned to a scope, and the size of the scope (including types of servers) needs to be monitored to make sure no schedule overrun occurs.

Global Technology Services

IBM

Scope Definition Layout for a Domain CMDB



- The 4 server scopes contain disjoint sets of servers, including unused IP addresses
- There is one load balancer scope that is executed 4x/day
  - Every load balancer is discovered and sync'ed on a 6-hour duty cycle
- Syncs happen 4 times a day, each containing a LB scope update and one server scope
  - Every server is discovered and sync'ed on a 24-hour duty cycle
- Each scope is manually defined → product enhancement needed for dynamic scopes

Slide 8 Best Practices for Deploying a CMDB |
IM 2009: 11th IFIP/IEEE International Symposium on Integrated Network Management

© 2009 IBM Corporation

Scope Definition is the process where we try to progressively and iteratively build a set of scope definitions against which a successful discovery can be initiated, while taking into account variables like adherence to schedule completion time, synchronization performance and type of infrastructure to be discovered. For the sake of our example, we assume that we need a total of five domain CMDBs to cover the whole range of enterprise IT assets.

The key is that separate scopes are defined for servers and load balancers. *While the same load balancer scope* (yellow color in the above figure) *will be executed every six hours*, *each of the four server scopes* (blue color) *contains a disjoint set of servers*. In other words, the overall server population of a data center is distributed across 4 disjoint scopes. Each of these 4 server scopes runs once a day, thus meeting the 24-hour SLA for servers. This ensures that every Domain CMDB meets the requirement; what is missing so far is the synchronization so that the updates are propagated to the Enterprise CMDB.

To accomplish this, each discovery step (run server and, subsequently, load balancer scopes) must be followed by an eCMDB synchronization. Since we are constrained by a 24-hour duty cycle, it is important to make sure that the dCMDBs are able to accomplish the scans and the synchronization within the stipulated timeframe. It is essential that this collective process (Discovery+Synchronization) be completed in less than 6 hours for every dCMDB that is a part of this setup. In particular:

- Every Server Scope discovery must finish in under 4 hours.
- Every Load Balancer Scope must finish in less than 1 hour.
- The synchronization process must complete in less than 1 hour.

This ideal scope layout was achieved by running discovery many times and iteratively building the right server scopes. The algorithm for creating and balancing the server scopes is described on the following slide 9. We observe that the domain CMDB is continuously running discoveries, which are followed by synchronizations with the eCMDB. In other words, the domain CMDB is almost never idle.

Assigning discovery profiles to scopes

A discovery profile is a list of sensors that will be executed on the discovered system in order to obtain its hardware and software configuration. Discovery profiles are defined for each scope. In order to prevent invocation of unneeded sensors in each scope, we create two different discovery profiles: one for the load balancer scope, another for the 4 server scopes. The former profile contains only the sensors that relate to load balancers, one of which being the SNMP sensor. The latter profile contains all the sensors for managed resource types (servers, storage, operating systems, middleware, applications etc.) that must be detected.

Algorithm for balancing Scopes

1. Run Initial Discovery
2. For each discovered subnet:
   1. Determine #servers within each subnet
   2. Classify servers by OS
   3. Count servers on a per-OS basis (in particular Windows)
3. Subnet_list := Sort subnets in descending order according to # of servers
4. Divide total amount of servers by # of scopes
   1. Yields threshhold of Windows/Unix servers
5. For each scope set:
   1. Determine for each OS if target # of servers > threshhold
   2. Move subnets between scopes so that each approximates threshhold
6. Run Discovery, determine median scope execution time
   1. if median > allowed_time_interval: error (too many servers, need additional dCMDB)
7. For each scope set:
   1. If scope execution time < allowed_time_interval: end (success)
   2. if actual_time => median, place subnets on unload_list
   3. if actual_time < median, obtain subnet(s) from unload_list
8. Goto 6

© 2009 IBM Corporation

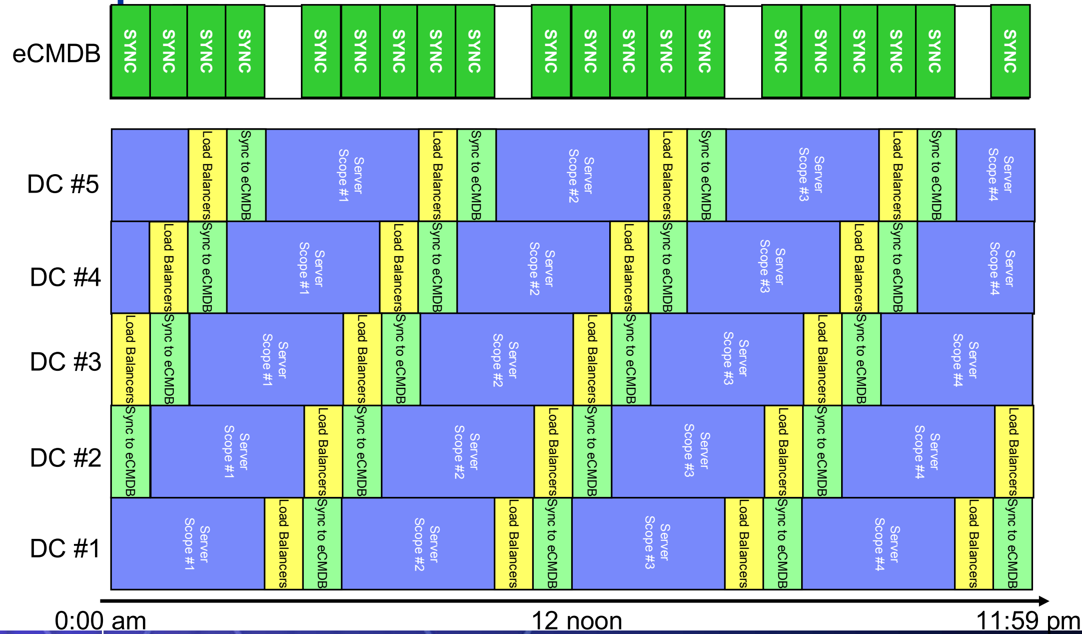
As mentioned before, the key to implementing different discovery cycles for different resource types in TADDM is to fill the allocated time intervals in a way that minimizes the likelihood of schedule overruns. The following algorithm, described in pseudo-code, is used to partition a given server population into scopes with minimal variability.

The first phase of the algorithm aims at identifying what resources are present in a given address range, and get a feeling how they are distributed across the various subnets. This is accomplished by a credential-less TADDM level 1 discovery, which first identifies resources on the network by a ping command, and subsequently attempting to determine the operating systems by means of their *nmap* [5] fingerprint. The results are being counted on a per-subnet basis and classified by their operating system (either 'Windows' and 'other'); the subnets are then ordered by the number systems they contain, in descending order. While this gives an accurate count of the 'live' systems on the network, the number of Windows servers is important sizing information for the Windows gateways that need to be deployed in order to perform deeper discoveries (OS level 2 and application level 3) subsequently. Recall that non-Windows systems are directly discovered by means of the discovery sensors running on the dCMDB. Subsequently, one divides the number of Unix/Linux and Windows systems, respectively by the amount of scopes that has been derived from the SLAs on the discovery cycles. This gives the average number of Unix and Windows systems that need to be discovered in each scope, which we need to target. We then iterate over the descending subnet list and move subnets across the scopes.

In a second phase of the algorithm, we need to fine-tune the initial distribution because we still may have large differences in terms of number of CIs between systems even if they run the same operating system. To accomplish this, we leverage the level 2 and level 3 discovery capabilities that require credentials and summarize the execution times of each discovery sensor on a per-machine basis. If we find that the median execution time of the scopes is above the time interval allotted to each scope, we need to deploy an additional domain CMDB because clearly because there is no solution to the server-to-scope allocation problem in the current configuration. We would then split the scope list into two parts, allocate each to a dCMDB and restart the algorithm. Otherwise, what remains to be done is the re-allocation of subnets between scopes until the variance in terms of discovery times is minimized.

The above steps need to be done over a period of time in order to ensure that the SLA pertaining to the discovery duty cycles are met. In addition, scope time overruns must be monitored so that changes in the data center do not lead to large variances between the scopes when old systems are retired and new systems are deployed, or subnets get (de)allocated.

Scope & Synchronization Plan for Domain & Enterprise CMDBs



© 2009 IBM Corporation

While the process of defining a scope layout for each dCMDB is quite complicated because of the numerous variables that need to be accounted for, the picture gets even more complex if we need to account for multiple dCMDBs that need to interact with an eCMDB. To ensure consistency in the data and the discovery runs, one needs to make sure that the scopes are staggered so that a discovery of resources will be followed by a synchronization to the eCMDB. Synchronization exports all the changes since the last sync from the dCMDB, sends them over the wire, and subsequently imports the changes into the eCMDB.

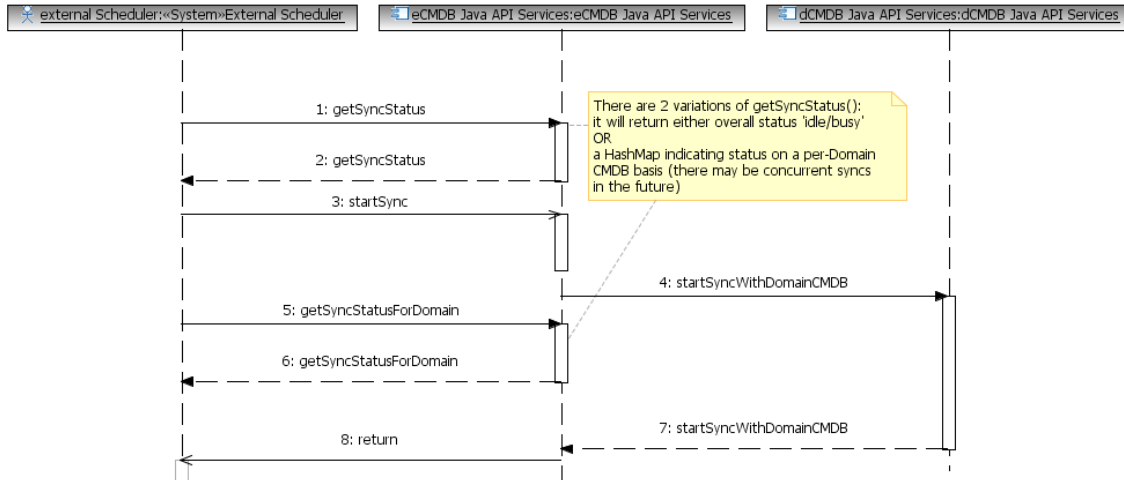
The above figure depicts the staggered scope layout from the previous slide when we apply it to the 5 Domain CMDBs we need to set up, one in each data center. The reason why the schedules of the Domain CMDBs are offset by an hour is because the eCMDB can only sync with one Domain CMDB at a time. One implication is that the Enterprise CMDB is running almost continuously (apart from four 1-hour breaks when no dCMDB is sync'ing); a second implication is that this scheme hits its peak when a 6th Domain CMDB is switched on, in case a large amount of additional servers needs to be accommodated. There would simply be no more room in the eCMDB schedule to accommodate any dCMDB beyond the 6th instance.

From the above figure, it is clear that managing a total of 60 discover and synchronize operations across 5 dCMDBs needs to be orchestrated by means of a scheduler that is external to the eCMDB and dCMDBs. In particular, the scheduler needs to keep track which dCMDB is running a discovery, and which dCMDB is synchronizing with the eCMDB at any given point in time. Based on this information, and by accounting for the task dependency chains '*server discovery* → *load balancer discovery* → *synchronize with eCMDB*' the scheduler invokes the necessary operations on all 5 dCMDBs (start discovery) and the eCMDB (initiate synchronization with dCMDB #x). To do so, the following commands must be accessible through an API:

- On the Domain CMDBs, the API needs to expose commands that allows the start and stop of every discovery, as well as a status inquiry to find out if the dCMDB is running a discovery.
- On the Enterprise CMDB, the API must expose a command that allows the start of the synchronization with a dCMDB, as well as a status inquiry that returns if the eCMDB is currently synchronizing, and, if yes, with which Domain CMDB.

While the former API existed in the product out-of-the-box, we designed the latter API and worked with the TADDM development team to incorporate it into the product. The details of this work are given on the following slide 11.

Initiating Synchronization from an External Scheduler



- Non-blocking Invocation of the eCMDB through both a Java API and shell script
- `api.sh -u username -p password sync <start|stop|status> -d <domainCmdbName>`

© 2009 IBM Corporation

As mentioned before, the Enterprise CMDB serves as the system of record. Data synchronization needs to be done from the dCMDB to the eCMDB on a regular basis to ensure that the data is current and valid. In other words, synchronization from a Domain CMDB to the Enterprise CMDB is a pull-based mechanism, initiated by the eCMDB or an external scheduler.

The sequence diagram highlighted above illustrates the synchronization process. The external scheduling system needs to the have the client APIs and the right set of credentials to initiate this session. This Java and command-line API can be invoked by any external system that has the right credentials.

The following steps are involved in controlling the synchronization actions:

1. The external scheduler connects to the eCMDB with the right set of credentials (not depicted in the figure)
2. The external scheduling system makes a call to the eCMDB to see if a synchronization is in process
3. If no synchronization is in progress, the scheduler issues a command to start the synchronization with a particular domain CMDB
4. Synchronization status is monitored periodically
5. When synchronization completes, the eCMDB polls the next dCMDB to start synchronization with it.

Recall that only one domain CMDB can synchronize with the eCMDB at a time. Also, if a new dCMDB is added to the system, it needs to be manually added to the eCMDB domain list.

Discovered Resources and Times for a Domain CMDB

- About 1000 Server Equivalents can be discovered on a 4-hour cycle
  - 4 x 4 hour cycles → 16 hours total
  - 4 x 1 hour for load balancer discovery
    - 62 load balancers discovered in 38 minutes
  - 4 x 1 hour for synchronization to eCMDB
- A sample Domain comprises a 64k IP address range (# live servers: 3,116)

Scope# / subnets	Duration	Server Type Details
12 x /21 subnets **8 x /24 subnets** **1 x /30 subnet**	**2:43 hours**	**394 Windows Servers** **433 Solaris and Linux Servers**
12 x /21 subnets **9 x /24 subnets**	**3:45 hours**	**389 Windows Servers** **333 Solaris and Linux Servers**
1 x /16 subnet **8 x /21 subnets** **1 x /23 subnet** **7 /24 subnets**	**3:50 hours**	**322 Windows Servers** **521 Solaris and Linux Servers**
2 x /21 subnets **6 x /24 subnets**	**3:30 hours**	**463 Windows Servers** **261 Solaris and Linux Servers**

© 2009 IBM Corporation

In order to evaluate whether the approach presented over the past few slides is feasible and accomplishes its goals, we will now provide sample data from one of the domain CMDBs to provide the reader with quantitative evidence. An enterprise data center contains a set of heterogeneous systems comprising servers, load balancers and network devices.

In our experiment, every domain CMDB was configured to discover a set of IP subnets totaling 64K IP addresses, which are split into smaller subnets of 2K, 4K, and 8K addresses. Each discovery scope was then built according to the algorithm described on slide 9; each scope contains a mix of 2K, 4K and 8K addresses. Configuring the system in this fashion ensured that the discovery times met the required SLAs. One can see that while 3 out of 4 scopes are relatively close in terms of duration, the first scope is faster. The reason why did not add additional systems from the other scopes is that we wanted to keep some room in case whole subnets with additional servers are deployed in the domain.
One may wonder why the discoveries start 'from scratch' every day by pinging potentially lightly populated IP subnets instead of leveraging prior knowledge (e.g., scanning only the topmost protocol layers of already discovered IP addresses). The reason is that we need to make sure that the discovery 1) picks up any newly installed system(s) and 2) flags (by means of an audit) what systems have recently gone offline for root cause identification and 3) make sure all systems connected to IP subnets are accounted for.

The following are additional observations:

1. The discovery times are higher whenever Windows Servers are part of the scope definitions. This is attributed to the fact that the discoveries of Windows servers is done via the Windows Gateways, which introduce additional delays. Also, a large amount of information is collected, resulting in the creation of about 450 CIs for every Windows server.
2. The discovery of Unix based systems is comparatively faster than Windows machines. Unix servers add about 150 CIs (Configuration Items) to the core tables of a dCMDB. The scan times are directly proportional to the number of CIs discovered.
3. The above table distinguishes only between operating systems. In the case of Solaris systems specifically, the above table is somewhat misleading because a Solaris system may be a 4-way rack-mounted x86-type server, or a large database server with more than 2000 NAS devices.
4. The Discovery times of Network Devices, especially Load Balancers, is dependent on the number of components that are attached / connected to them.

We have described best practices for deploying a Configuration Management Database (CMDB), which we have developed from several past client engagements. We found that while such best practices are crucial to ensure the success of the engagement, they need to be complemented by a good deal of 'social engineering': We found that in particular after the initial discovery, the customer may see for the first time what the environment really looks like – and also what may be wrong with it! The above slide lists several concrete examples where the CMDB classified systems differently than expected by the administrators; each of these cases were initially flagged as product issues, but turned out later on to be genuine anomalies. In addition, the customer's processes for obtaining credentials to explore the systems may be suboptimal and may add several weeks to the duration of an engagement.

There are a variety of areas for further research in the area of CMDB architecture and deployment: Some, like the need for developing a strategy of moving data from the 'live' CMDB into data warehouses, may become apparent only long after the project team has handed over the running system to the customer. Others, like improving the current issues with transactional consistency (highlighted on slide 6) will have direct positive impact on the scalability of the system. Finally, the earliest opportunity for a long-lasting impact (good or bad) on the project is the initial sizing of the environment to determine the type and amount of equipment that needs to be purchased. Improving techniques for modeling and estimating the performance of the CMDB will significantly help improve the time-to-market for a CMDB project, and thus its profitability.

References

[1] *IT Infrastructure Library: Service Transition*, Office of Government Commerce and TSO, 2007

[2] M. Burgess. Cfengine: A site configuration engine. *Computing Systems*, 8(3), 1995. USENIX Association, see also: http://www.cfengine.org.

[3] Y. Diao et al., Service Level Management: A dynamic Discovery and Optimization Approach, *IEEE Transactions on Network and Service Management*, Volume 1, Number 2, IEEE, December, 2004

[4] Gucer, V. et al., *Deployment Guide Series: IBM Tivoli Application Dependency Discovery Manager* V7.1, IBM Redbook sg247616, August, 2008

[5] *nmap – Network Mapper: Free Security Scanner For Network Exploration*. http://nmap.org. (as of February 2009)