SLA-Based SAN Design

Eray Gençay*[§], Carsten Sinz* and Wolfgang Küchlin*
* WSI for Computer Science, University of Tübingen, D-72076 Tübingen, Germany,
Email: see http://www-sr.informatik.uni-tuebingen.de/pages/staff.html
[§] IBM Deutschland GmbH, D-55131 Mainz, Germany, Email: egencay@de.ibm.com

***Abstract*—Storage Area Networks (SANs) connect storage devices to servers over fast network interconnects. We consider the problem of optimal SAN configuration with the goal of retaining more security in meeting service level agreements (SLAs) on unexpected peaks.**

First, we give an algorithm for assigning storage devices to applications running on the SAN's hosts. This algorithm tries to balance the workload as evenly as possible over all storage devices. Our second algorithm takes these assignments and computes the interconnections (data paths) that are necessary to achieve the desired configuration while respecting redundancy (safety) requirements in the SLAs. Again, this algorithm tries to balance the workload of all connections and devices. Thus, our network configurations respect all SLAs and provide flexibility for future changes by avoiding bottlenecks on storage devices or switches.

We also discuss integrating our solution with the open source SAN management software Aperi.

I. Introduction

Today's mainframe computers or servers do not contain disks for mass storage. Instead, business critical data is centrally stored in dedicated large capacity storage devices, which are connected to the host computers (servers) over fast interconnects like fibre channel switches and hubs. Such a Storage Area Network (SAN) may consist of dozens or even hundreds of hosts, switches, and storage devices, and therefore the proper design and management of a SAN is a non-trivial, but business critical, task.

In addition, SAN configurations have to fulfill Service Level Agreements (SLAs), which express performance requirements the SAN has to attain. For example, an SLA may guarantee certain throughput rates or storage capacities for a host application. Many SLAs reflect data flow requirements that originate from the applications running on the servers. While designing a SAN, these requirements should be considered besides "best-practices" constraints like redundancy rules, and technical constraints like the limits of the resources, e. g. the number of ports of a device.

Our rule-based system SANchk [2] already checks whether a given SAN configuration respects a number of best practices rules. In this paper, we consider the problem of configuring a SAN in an optimal way, while additionally taking a number of SLAs into account. Our primary concern is not to minimize hardware cost but to maximize the flexibility to accommodate changing SLA requirements in the future. This is because the most critical cost factor associated with a SAN is not raw hardware but operation downtime, e.g. for reconfiguration. In fact, downtime of business critical SANs is simply not an option beyond maybe one hour of scheduled maintenance once a year. Whenever a new or changed SLA cannot be accommodated because reconfiguration is too costly, it must be accommodated by purchasing additional new hardware. Therefore it is of prime importance for a new configuration to avoid future performance bottlenecks which may necessitate major reconfigurations.

For a given set of hardware devices, we attempt to achieve a uniformly high proportion of free resources at each device, grouped by device types (e.g., X% free storage capacity on each storage device and Y% unused ports in each switch). In this way, we can increase the probability that the QoS requirements of all applications are still met, even if some of the applications demand more resources than planned, because bottlenecks are avoided. This in turn will result in a decrease of SLA violations and SLA penalties, respectively.

The rest of the paper is organized as follows: Section II gives a formal definition of the first part of our SAN design problem, namely the problem of assignments of applications to storage devices, and demonstrates it with an example. Section III defines the second problem, the connection problem. Section IV discusses the integration of the solution into a SAN management framework on the example of Aperi. In Section V, we present work related to our paper. Finally, we draw our conclusions and consider the future work in Section VI.

II. SAN Storage Assignment Problem

In the storage assignment problem, we have as input the applications with their requirements (throughput and storage space), the information which host is serving which applications, and data about the storage devices' capacities (throughput, storage space). We want to find out, which applications should use which storage devices in order to get a distribution of the workloads on the devices as evenly as possible.

A. Assignment of Applications to Storage Devices

Let $D = \{d_1, \ldots, d_n\}$ be the set of storage devices, and $H = \{h_1, \ldots, h_m\}$ be the set of hosts. Storage devices $d_i \in D$ are modelled as pairs $d_i = (c_i, t_i)$, where c_i stands for the provided storage capacity and t_i for the provided throughput capacity of device i. In analogy, for all hosts $h_i \in H$, h_i is a pair $h_i = (c'_i, t'_i)$ consisting of the need for storage capacity c'_i of host i and its throughput requirement t'_i. To express that there is an assignment between a host and a storage device, we define a set $X = \{x_{i,j} \in \{0, 1\} \mid 1 \leq i \leq m, 1 \leq j \leq n\}$

with $x_{i,j} = 1$ iff there is a host i that is assigned to storage device j.

1) Applications and hosts: Let $A_i = \{a_1, \ldots, a_k\}$ be the set of applications that run on host i. Applications a are pairs (c', t') consisting of storage capacity requirement and throughput requirement. In our formalization, instead of mapping applications to storage devices, we use a simplifying trick: We represent each of the applications on a host as a "pseudo host". Thus, if there are k applications running on host i, we replace host i by k pseudo hosts. By doing this, we just have to map (pseudo) hosts to storage devices. In order to make this simplification step work, we have to conduct an additional trivial pre-check to see if the host can serve all the applications running on it with the desired throughput capacity. Our optimization then produces an assignment of pseudo hosts to storage devices. Since we know which application is running on which host, we can find out in a next step, which physical links are needed between hosts and storage devices.

In our approach, we define an application as an atomic entity that occupies an indivisible block on the storage device. Hence, we would represent a DBMS that uses different storage spaces for its table data and logging information—having different requirements for each of them—as two different applications in our model.

B. Constraint Blocks

In the following, we formulate some constraints (in Pseudo-Boolean logic) to ensure that we obtain valid assignments to the variables in the set X. We use $I_H = \{1, \ldots, m\}$ and $I_D = \{1, \ldots, n\}$ as index sets for hosts and storage devices, respectively.

1) Exactly one connection to a storage device for each pseudo host: Every pseudo host (application) should be served by exactly one storage device at the end of the computation: $\forall i \in I_H : \sum_{k=1}^{n} x_{i,k} = 1$.

2) Capacity constraints: It has to be ensured that the storage capacities of the storage devices are not exceeded: $\forall j \in I_D : \sum_{k=1}^{m} c'_k x_{k,j} \leq c_j$, i.e., the sum of the storage space requirements of all hosts that are assigned to a storage device d_j does not exceed the storage capacity c_j of the device.

3) Throughput constraints: This constraint block ensures that the port speeds provided by the storage devices are not exceeded: $\forall j \in I_D : \sum_{k=1}^{m} t'_k x_{k,j} \leq t_j$.

C. Optimization Problem

So far, we can find valid assignments between hosts and storage devices. The next step is the definition of a goal function for the optimization. We want the workloads for storage devices to be balanced as much as possible, so that the free resources on devices are maximized proportional to their capacities. This kind of optimization has advantages like the increase of flexibility in the choise of resources, since fewer devices would be working with their full capacity. Another advantage is that the probability of violating service level agreements would also be decreased. Since we maximize the unused part of the devices' resources, it is less likely that the SLAs are violated, even if some of the applications behave unexpectedly.

1) Scaling of inequalities: In order to achieve an equal balancing of the throughputs, we first have to scale all throughput constraints such that their right hand sides become equal. By doing this, we can generate constraints that limit the throughput for each storage device to a constant factor below what is maximally possible. Scaling also serves to obtain integer coefficients, as in our case the constraint solver can only handle such constraints.

Using scaling factors s_j^* $(1 \leq j \leq n)$ for our throughput constraints, we obtain: $\forall j \in I_D : s_j^* \cdot \sum_{k=1}^{m} t'_k x_{k,j} \leq s_j^* \cdot t_j$. We define the scaling factors by $s_j^* = 1/t_j \cdot \prod_{i=1}^{n} t_i$ to achieve equal right hand sides $t^*_{\text{norm}} = \prod_{i=1}^{n} t_i$ of the throughput inequalities. We might need an additional scaling factor s_I (now the same for all inequalities) to convert the coefficients to integers, but we will not consider this in our further discussion, and assume that after scaling with the factors s_j^* all coefficients are integers.

Obviously, one would like to make the number t^*_{norm} as small as possible in order to reduce the time the solver needs to process the problem. To achieve this, we divide all inequalities by the greatest common divisor q of all occurring coefficients. We call the resulting scaling factors s_j, i.e. $s_j = s_j^*/q$. Similarly, we call the resulting common right hand side of the inequalities t_{norm}, i.e. $t_{\text{norm}} = t^*_{\text{norm}}/q$.

2) Auxiliary variables: To transform the satisfiability problem we have obtained so far into an optimization problem, we introduce a set of auxiliary variables $L = \{l_0, \ldots, l_p\}$. The auxiliary variables are the binary representation of a number $l = l_0 + 2 \cdot l_1 + \cdots + 2^p \cdot l_p$. These auxiliary variables are added to the scaled throughput capacity inequalities: $\forall j \in I_D : s_j \cdot \sum_{k=1}^{m} t'_k x_{k,j} + \sum_{r=0}^{p} 2^r l_r \leq t_{\text{norm}}$.

The auxiliary variables in an inequality symbolize the unused resources on the device. The idea now is that we maximize the value of l. Since $l_i \in \{0, 1\}$ for all auxiliary variables $l_i \in L$, we still have a Pseudo-Boolean problem.

3) Objective function: The objective function of the optimization problem maximizes the sum of the auxiliary variables, formally: $\max \sum_{r=0}^{p} 2^r l_r$.

D. Test of Preconditions

Before generating the constraint system, we check some trivial preconditions, whose unsatisfiability implies the unsatisfiability of the whole constraint system due to an invalid configuration of hosts, applications or storage devices.

1) Throughput requirements of the applications are too high: If the sum of the throughput requirements of the applications that are resident on a host is greater than the throughput rate that is supplied by the host bus adapter of that host, then it is impossible to find a satisfying solution. Formally: Let $T_{A_i} = \{t_1, \ldots, t_s\}$ be the set of the throughput requirements of the applications on host i and $t_{\text{sum},i}$ the sum of the throughput rates of the ports of the host. If we have $\sum_{t_i \in T_{A_i}} t_i > t_{\text{sum},i}$, the configuration is invalid.

2) Storage space requirements of applications are too high: If the storage space requirement of a single host or a single application is greater than the greatest available capacity on the side of the storage devices, the configuration is unsatisfiable. Formally: Let C_{A_i} be in analogy to 1) the set of storage space requirements of the applications that are resident on host i, and $c_{\max}$ the maximal storage capacity among the storage devices. Then the configuration is unsatisfiable, if we have $\exists c_k \in C_{A_i}, i \in I_H : c_k > c_{\max}$.

E. Example

In the following, we demonstrate a small example with three applications and three storage devices. $A = \{a_1, a_2, a_3\}$ is the application set, where $a_i = (t'_i, c'_i)$ is a pair consisting of the throughput and capacity requirements for the application. The applications are supposed to have the following settings: $t'_1 = 0.1$, $t'_2 = 1.0$, $t'_3 = 0.5$, $c'_1 = 50$, $c'_2 = 100$, $c'_3 = 70$.

The storage devices are represented as a set $D = \{d_1, d_2, d_3\}$ that contains tuples with the provided throughput and storage capacities of the storage devices: $d_i = (t_i, c_i)$. The storage devices have the following settings: $t_1 = 2.0$, $t_2 = 1.0$, $t_3 = 1.0$, $c_1 = 200$, $c_2 = 120$, $c_3 = 300$.

Now, the constraints are as follows:

1) Exactly one connection to a storage device for each pseudo host: $x_{1,1} + x_{1,2} + x_{1,3} = 1$, $x_{2,1} + x_{2,2} + x_{2,3} = 1$, $x_{3,1} + x_{3,2} + x_{3,3} = 1$.

2) Capacity constraints: $50 \cdot x_{1,1} + 100 \cdot x_{2,1} + 70 \cdot x_{3,1} \leq 200, 50 \cdot x_{1,2} + 100 \cdot x_{2,2} + 70 \cdot x_{3,2} \leq 120, 50 \cdot x_{1,3} + 100 \cdot x_{2,3} + 70 \cdot x_{3,3} \leq 300$.

3) Throughput constraints: $0.1 \cdot x_{1,1} + 1 \cdot x_{2,1} + 0.5 \cdot x_{3,1} \leq 2, 0.1 \cdot x_{1,2} + 1 \cdot x_{2,2} + 0.5 \cdot x_{3,2} \leq 1, 0.1 \cdot x_{1,3} + 1 \cdot x_{2,3} + 0.5 \cdot x_{3,3} \leq 1$.

4) Optimization function: To build the optimization function, we must first scale the throughput constraints to obtain a common right hand side and integer coefficients (in our example, we have to scale the constraints to a common right hand side of 20). Then, we calculate how many auxiliary variables we need, and add them with their corresponding coefficients to every single throughput constraint. The optimization function is to maximize this sum. To cover a range of 0 to 20, we need five auxiliary variables, and thus $l = l_0 + 2l_1 + 4l_2 + 8l_3 + 16l_4 + 32l_5$.

After getting the problem solved by the solver OPBDP [1], an implementation of an implicit enumeration algorithm for solving (non-)linear Pseudo-Boolean optimization problems with integer coefficients, the following variables were fixed to 1: l_1, l_3, $x_{1,2}$, $x_{2,1}$, $x_{3,3}$. The resulting SAN storage assignment is shown below.

F. Empirical Results

In our initial tests, it took less than one minute to calculate problems of size up to 15 hosts and 15 storage devices with real device attribute data and manually generated QoS requirements information. From size 16 x 16 on, the measured run times were at least 10 minutes. Our test results are shown in Table I.

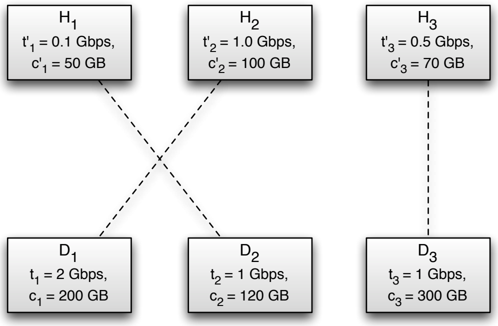

Fig. 1. An example configuration with three hosts and three storage devices, and their computed assignments.

TABLE I
SOLVING TIMES FOR PROBLEMS UP TO SIZE 15 X 15.

Hosts	Storage Devices	Mean Time (s)
10	10	0.3
11	11	0.3
12	12	0.5
13	13	14.3
14	14	12.7
15	15	51.2

III. SAN CONNECTION PROBLEM

The second SAN configuration problem we consider deals with how to lay out connections in a SAN, which link hosts, switches, and storage devices. Certain criteria (SLAs) have to be met in order to obtain a correct configuration. For example, it is typically required that all paths from hosts to storage devices are redundant.

We assume that the assignment from hosts to storage devices is already given (e.g., computed by the algorithm we have given in Section II), i.e. we already know which paths are needed and their required throughput.

We assume sets of hosts $H = \{h_1, \ldots, h_k\}$, switches $S = \{s_1, \ldots, s_l\}$ and storage devices $D = \{d_1, \ldots, d_m\}$. By $\mathcal{D} = H \cup S \cup D$ we denote the set of all SAN devices. Moreover, we assume a set of paths $P = \{p_1, \ldots, p_n\}$ which are to be routed through the SAN. Each path $p \in P$ connects a host $h(p) \in H$ to a storage device $d(p) \in D$, but the individual hops are not known. Redundancy is modeled by having the same host and storage device connected by more than one path. By $\mathrm{thr}(p)$ we denote the required throughput of a path $p \in P$. Similarly, $\mathrm{thr}(x)$ denotes the maximal throughput for a single port of device $x \in \mathcal{D}$ (we assume the throughput to be the same for all ports of a device). By $\mathrm{ports}(x)$ we denote the number of available ports on device x. To characterize a path, we use predicates $\mathrm{conn}(x, y, p)$, denoting that device x is (directly) connected to device y on path p, and functions $\mathrm{mult}(x, y, p)$ to denote the required multiplicity of the link between x and y on path p, if it is realized[1]. Note that $\mathrm{mult}(x,y,p)$ can be computed in advance by $\mathrm{mult}(x,y,p) = \lceil \mathrm{thr}(p) / \min\{\mathrm{thr}(x), \mathrm{thr}(y)\} \rceil$. We will also use auxiliary predicates $x \in p$ to denote that device x occurs on path p. Note that $\mathrm{conn}(x,y,p)$ implies $x \in p \wedge y \in p$.

Now we can specify the constraints for a correctly configured SAN network:

1) Each $p \in P$ must be a valid path in the network, connecting device $h(p)$ with device $d(p)$, i.e. there must be a sequence $s_1, \dots, s_t$ of switches, such that the predicates $\mathrm{conn}(h(p), s_1, p)$, $\mathrm{conn}(s_t, d(p), p)$, and $\mathrm{conn}(s_i, s_{i+1}, p)$ for all $1 \le i < t$ hold.
2) Redundant paths must not use the same switches. I.e., if paths p_1 and p_2 are redundant—which we will denote by $\mathrm{red}(p_1, p_2)$—then $s \in p_1$ implies $s \notin p_2$ for all $s \in S$.
3) For each path p, the throughput requirement must be satisfied, i.e. $\mathrm{mult}(x,y,p) \cdot \min\{\mathrm{thr}(x), \mathrm{thr}(y)\} \ge \mathrm{thr}(p)$ must hold for all x, y, for which $\mathrm{conn}(x,y,p)$ is true. This constraint always holds if we use the definition for $\mathrm{mult}(x,y,p)$ as given above.
4) The number of ports of each device must be sufficient: $\sum_{p \in P, y \in \mathrm{out}(x), z \in \mathrm{in}(x)} \mathrm{mult}(x,y,p) + \mathrm{mult}(z,x,p) \le \mathrm{ports}(x)$ for all $x \in \mathcal{D}$, where $\mathrm{out}(x) = \{u \in \mathcal{D} \mid \mathrm{conn}(x,u,p)\}$ and $\mathrm{in}(x) = \{u \in \mathcal{D} \mid \mathrm{conn}(u,x,p)\}$.

As our optimization goal we have chosen to minimize the fraction of ports that are used on each device. This goal ascertains that the load on all devices is equally balanced. It can be expressed as $\min \max_{x \in \mathcal{D}} \left\{ \frac{\mathrm{ports\_used}(x)}{\mathrm{ports}(x)} \right\}$, where $\mathrm{ports\_used}(x)$ is the expression on the left hand side of the inequality in constraint 4. To convert this expression to a Pseudo-Boolean optimization goal, we use the same scaling trick that we already used in Section II-C, i.e. we scale the inequalities in constraint 4 such that they possess a common right hand side, and then introduce a slack variable $l = \sum_{i=0}^{p} 2^i l_i$ on the left hand side of each inequality. Afterwards we maximize the slack variable. We want to conclude this section with two remarks: First, the maximal path length t (in number of switches) is typically quite low, e.g. 3, which facilitates the encoding of paths. And second, SAN devices often put a limit on the maximal trunk width (i.e. the number of "parallel" links between two devices). This may be specified by an additional restriction similar to constraint 4. All these constraints can be converted to Pseudo-Boolean logic and handled by a standard Pseudo-Boolean solver.

A. Encoding as a Pseudo-Boolean Problem

We will now show in detail how the constraints given in the last section can be encoded as a Pseudo-Boolean optimization problem. First, we encode paths: Let $t_{\max}$ be the maximal number of switches that shall occur on a path. We encode each path p as a sequence $(u_p, v_{p,1}, \dots, v_{p,t_{\max}}, w_p)$ of binary numbers, where $v_{p,i}$ is the number of the i-th switch on path p $(1 \le v_{p,i} \le l)$, $u_p + 1$ is the index of the host $(0 \le u_p < k)$ and $w_p + 1$ is the number of the storage device $(0 \le w_p < m)$ of path p. A value $v_{p,i}$ of zero indicates the end of a path. For each path we thus need $t_{\max} \cdot \mathrm{ld}(l+1) + \mathrm{ld}(k) + \mathrm{ld}(m)$ bits to encode it.[2] Note that the values of u_p and w_p are fixed by the SAN specification, whereas the values of the $v_{p,i}$ are undetermined. We derive predicates $\mathrm{red}(p_1, p_2)$ and ports_suff as follows:

$$\mathrm{red}(p_1,p_2) \Leftrightarrow \bigwedge_{1 \le i \le t_{\max}} \Big(v_{p_1,i} \neq 0 \Rightarrow \bigwedge_{1 \le j \le t_{\max}} (v_{p_2,j} \neq 0 \Rightarrow v_{p_1,i} \neq v_{p_2,j}) \Big)$$

$$\text{ports\_suff} \Leftrightarrow \bigwedge_{1 \le i \le k} \text{ports\_suff}_{\mathrm{H}}(h_i) \wedge \bigwedge_{1 \le i \le l} \text{ports\_suff}_{\mathrm{S}}(s_i) \wedge \bigwedge_{1 \le i \le m} \text{ports\_suff}_{\mathrm{D}}(d_i)$$

$$\text{ports\_suff}_{\mathrm{H}}(h) \Leftrightarrow \sum_{\substack{p \in P, s \in S \\ h(p) = h}} \mathrm{mult}(h,s,p) \cdot (v_{p,1} = s) \le \mathrm{ports}(h)$$

$$\begin{aligned} \text{ports\_suff}_{\mathrm{S}}(s) \Leftrightarrow \sum_{p \in P} \Big(& \mathrm{mult}(h(p),s,p) \cdot (v_{p,1} = s) \\ & + \sum_{s' \in S, s' \neq s} \big(\mathrm{mult}(s,s',p) \cdot \textstyle\bigvee_{1 \le i < t_{\max}} (v_{p,i} = s \wedge v_{p,i+1} = s') \\ & + \mathrm{mult}(s',s,p) \cdot \textstyle\bigvee_{1 < i \le t_{\max}} (v_{p,i} = s \wedge v_{p,i-1} = s') \big) \\ & + \mathrm{mult}(s,d(p),p) \cdot L(s,p) \Big) \le \mathrm{ports}(s) \end{aligned}$$

$$\text{ports\_suff}_{\mathrm{D}}(d) \Leftrightarrow \sum_{\substack{p \in P, s \in S \\ d(p) = d}} \mathrm{mult}(s,d,p) \cdot L(s,p) \le \mathrm{ports}(d)$$

where $L(s,p)$ is true, if s is the last switch on path p:

$$L(s,p) \Leftrightarrow \bigvee_{1 \le t \le t_{\max}} \Big((v_{p,t} = s) \wedge \bigwedge_{t < t' \le t_{\max}} (v_{p,t'} = 0) \Big)$$

Moreover, we have to restrict the values $v_{p,i}$ to admissible values in the range $[0, l]$, as long as $l+1$ is not a power of 2. This can be achieved easily by further Boolean logic constraints.

The equations $v_{p,i} = x$ are translated as follows:

$$v_{p,i} = x \quad \Leftrightarrow \quad \bigwedge_j \big(\mathrm{bit}_j(v_{p,i}) \Leftrightarrow \mathrm{bit}_j(x) \big)$$

(with obvious simplifications if x is a constant), where $\mathrm{bit}_j(x)$ denotes the value of bit j in the binary representation of x.

As a further constraint, we might add that on each path at least one switch is used. This excludes degenerate cases, where hosts are directly connected to storage devices, which is typically not considered as a good SAN design. To achieve this, we simply add the constraint $v_{p,1} \neq 0$ for each path p.

The whole set of Pseudo-Boolean constraints for the SAN connection problem then consists of the set of redundancy

[1]We use the predicates $\mathrm{conn}(x,y,p)$ always in such a way that x is closer to the "host side" and y is closer to the "storage device side".

[2]By $\mathrm{ld}(x)$ we denote the *logarithmus dualis* of x.

constraints $\mathrm{red}(p_1, p_2)$ for all paths that should be redundant, plus the constraints generated by $\mathrm{ports\_suff}$, plus the additional definitions for the $v_{p,i}$, $L(s,p)$, and C_k.

Our encoding as given does not allow sharing of links, so far. However, by changing the definition of $\mathrm{mult}(x, y, p)$, such that it would also allow "fractions of ports" to be allocated, sharing of links could be formalized.

Another optimization goal, slightly different from the one given above, would be to balance the throughput of the used ports of each switch as evenly as possible. By this, smaller changes in the throughput requirements would not lead to the necessity of reconfiguring the SAN. Such an optimization goal could also be realized by allowing fractions of ports to be virtually allocated.

IV. Implementation in a SAN Management Framework

Since SANs are managed mostly by an integrated and central management software, it is very likely that such a solution is to be implemented as part of a storage management software. An example of such a software framework is Aperi. Aperi is an open source project at the Eclipse Foundation. The project aims to provide a framework for SAN management based on open standards. In an earlier work, we integrated a solution (SANchk) for SAN configuration checking according to "best practices rules" into Aperi [2], [3]. An optimization tool with the function described above can be integrated into Aperi as plugins in analogy to that.

Aperi is based on Equinox and the Rich Client Platform. Equinox is the reference implementation of the "R4 core framework specification" from the standard Open Services Gateway initiative (OSGi). Using Equinox, plugins can be directly integrated into Eclipse and thus also into Aperi to extend the framework.

The major components of Aperi are two servers (data server and device server), an RCP-GUI, a database, and agents that run on hosts and collect data. Any extension to Aperi should extend some of these components. Since Aperi uses the OSGi framework, it provides for each component its own Extension Points.

Aperi has a request-response architecture, which uses Service Provider, Request Handler, Request and Response objects. An extention to the framework should also use these objects to accomplish the communication between the new plugin or plugins and the components of Aperi.

In Figure 2, we demonstrate, how the solution could be integrated into a storage management framework on the example of Aperi with our configuration checking plugin SANchk. Except for some small improvements, Aperi and SANchk do not need to be modified. In order to enable SLA-based network design optimization, Aperi database should be extended by tables for applications that are running on the hosts and their SLA requirements. On the SANchk side, attributes to the XML elements for rules and their rule parameters should be added, which indicate how a rule should be transformed into a constraint in OPB format.

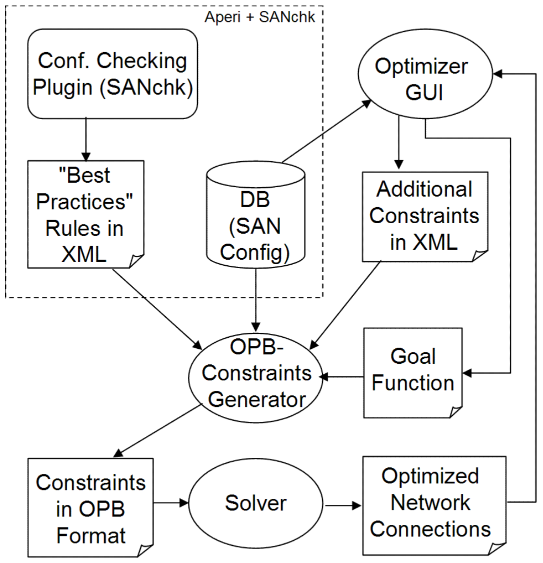
Aperi + SANchk
Conf. Checking Plugin (SANchk)
"Best Practices" Rules in XML
DB (SAN Config)
Optimizer GUI
Additional Constraints in XML
OPB-Constraints Generator
Goal Function
Constraints in OPB Format
Solver
Optimized Network Connections

Fig. 2. Integration of the optimizer into Aperi.

Optimizer GUI should help users add some custom constraints, modify settings and see results after computation. The GUI should show the current network design at the start, so that users can exclude one part of the devices from the optimization, or can append a fictitious device that is planned to be procured. These actions on the GUI can than be transformed into constraints in OPB format and added to the constraint system. The goal function for the constraint system should also be specified on the GUI. Users should be able to choose among several different optimization goals.

OPB Constraints Generator obtains input from several sources. It uses rules from the configuration checking plugin that are relevant for network design like redundancy rules, additional constraints that are generated by the GUI as mentioned above, the chosen goal function, and the information about the network components (existing hosts, storage devices, switches etc. with their capacities) to generate single OPB constraints. After the constraints in OPB format are generated, the problem should be solved by an off-the-shelf solver. The solver should have an API in Java or C in order to be able to work with Aperi. After the computation, the results should be reported to users on the GUI.

The functionality described above can be packed into two plugins, one for the GUI and the other for the application logic. The plugin for application logic should contain the OPB Constraints Generator and the code that handles the solver. It should also contain a Request Handler in order to accept Request objects from the GUI plugin. After processing the request, it should send back a Response object. The GUI plugin should contain beside the Optimizer GUI components also an extra Eclipse view and other auxiliary classes that are needed for the integration of the GUI into the Aperi GUI.

V. Related Work

This paper is an extended version of the short paper [4]. Here, we additionally discuss the integration of the solution into the open source SAN management software Aperi, present complete encodings for the algorithms, and give preliminary test results for the first optimization problem.

Optimization problems related to SAN design have been treated previously in several publications ([5], [6], [7], [8], [9], [10]). They differ from our approach mainly in the definition of the optimization problem and its goal function, in the input parameters or methods used to solve the problem.

Ward et al. [5] present two different heuristic algorithms, namely FlowMerge and QuickBuilder, to design a SAN automatically. FlowMerge merges single flows of data that share a switch or hub in a set of flows in a recursive way. Beginning with a fully bipartite, directed graph (every host is connected with every storage device), the number of the edges are decreased incrementally while hub and switch nodes are added to the network. Dicke et al. ([6], [7]) use biologically inspired approaches to improve SAN designs or to create new SAN designs. Walker et al. [11] have a mixed-integer approach to the SAN design problem and also use the minimal provisioning cost as the optimization goal. They focus on the Core-Edge reference topology and provide two formulations for the SAN design problem. A framework for automated storage management based on QoS specifications is Rome [12] by HP Laboratories. Singh et al. [**?**] have proposed a SAN FS planning tool that uses the notions of application templates and a planning engine to assist a system designer with the design process. PulsatingStore [13] is an analytical framework that provides an automated storage management service for DBMS that balances the conflicting goals of performance guarantees and on-demand resource usage. Anderson et al. [14] present the Disk Array Designer (DAD) that uses a generalized best-fit bin packing heuristic to design disk arrays according to both capacity and I/O performance demands for the application data. Reiss and Kanungo [15] examine the problem of choosing a QoS level for each table or index in a service provider's backend databases to minimize the cost of provisioning storage while satisfying the application level SLAs.

VI. Conclusion

We defined and formalized the SAN design problem to increase the flexibility of a SAN instead of to minimize the provisioning cost. The flexibility increases on the one hand its ability to meet the QoS requirements for the applications running on its hosts, even if some of the workloads behave themselves spontaneously irregularly, and on the other hand the ability of the network to grow without the need of major structural changes.

We discussed the implementation of the solution in a storage management framework on the example of the Open Source project Aperi.

Acknowledgment

The research presented in this paper was funded by IBM Deutschland GmbH. We are grateful to the storage experts at IBM Deutschland GmbH, Mainz for their technical support in storage related subjects.

References

[1] E. Gençay, W. Küchlin, and T. Schäfer, "SANchk: An SQL-based validation system for SAN configuration," in *IEEE/IFIP Symposium on Integrated Network Management, IM 2007*, 2007, pp. 333–342.

[2] P. Barth, "A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization," Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, Research Report MPI-I-95-2-003, January 1995.

[3] E. Gençay, C. Sinz, W. Küchlin, and T. Schäfer, "SANchk: SQL-based SAN configuration checking," *IEEE Transactions on Network and Service Management*, 2008, conditionally accepted. [Online]. Available: http://www-sr.informatik.uni-tuebingen.de/~gencay/tnsm_gencay.html

[4] E. Gençay, C. Sinz, and W. Küchlin, "Towards SLA-based optimal workload distribution in SANs," in *IEEE/IFIP Network Operations and Management Symposium, NOMS 2008*, 2008. [Online]. Available: http://www-sr.informatik.uni-tuebingen.de/~gencay/noms2008_gencay.html

[5] J. Ward, M. O'Sullivan, T. Shahoumian, and J. Wilkes, "Appia: Automatic storage area network fabric design," in *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2002, p. 15.

[6] E. Dicke, A. Byde, P. J. Layzell, and D. Cliff, "Using a genetic algorithm to design and improve storage area network architectures." in *GECCO (1)*, 2004, pp. 1066–1077.

[7] E. Dicke, A. Byde, D. Cliff, and P. J. Layzell, "An ant inspired technique for storage area network design," in *BioADIT*, 2004, pp. 364–379.

[8] S. Uttamchandani, G. A. Alvarez, and G. Agha, "DecisionQoS: An adaptive, self-evolving QoS arbitration module for storage systems," in *POLICY*, 2004, pp. 67–76.

[9] S. Uttamchandani, L. Yin, G. A. Alvarez, J. Palmer, and G. Agha, "CHAMELEON: a self-evolving, fully-adaptive resource arbitrator for storage systems," in *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2005, pp. 6–6.

[10] S. Uttamchandani, K. Voruganti, S. Srinivasan, J. Palmer, and D. Pease, "Polus: Growing storage QoS management beyond a "4-year old kid"," in *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2004, pp. 31–44.

[11] C. Walker, M. O'Sullivan, and T. Thompson, "A mixed-integer approach to core-edge design of storage area networks," *Comput. Oper. Res.*, vol. 34, no. 10, pp. 2976–3000, 2007.

[12] J. Wilkes, "Traveling to Rome: QoS specifications for automated storage system management," in *IWQoS '01: Proceedings of the 9th International Workshop on Quality of Service*. London, UK: Springer-Verlag, 2001, pp. 75–91.

[13] A. Singh, K. Voruganti, S. Gopisetty, A. Fleshler, R. Routray, and C. hao Tan, "SANFS Maestro: Resource planning for enterprise storage area network (SAN) file systems," in *CSREA EEE*, 2005, pp. 32–38.

[14] L. Qiao, D. Agrawal, A. E. Abbadi, and B. R. Iyer, "PULSATING-STORE: An analytic framework for automated storage management," in *ICDEW '05: Proceedings of the 21st International Conference on Data Engineering Workshops*. Washington, DC, USA: IEEE Computer Society, 2005, p. 1213.

[15] E. Anderson, S. Spence, R. Swaminathan, M. Kallahalla, and Q. Wang, "Quickly finding near-optimal storage designs," *ACM Trans. Comput. Syst.*, vol. 23, no. 4, pp. 337–374, 2005.

[16] F. R. Reiss and T. Kanungo, "Satisfying database service level agreements while minimizing cost through storage QoS," in *SCC '05: Proceedings of the 2005 IEEE International Conference on Services Computing*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 13–21.