TOWARDS TRANSACTIONAL INTEGRITY ISSUES IN POLICY BASED NETWORK MANAGEMENT SYSTEMS

Vitor Roque

Instituto Politécnico da Guarda, ESTG, Guarda – Portugal

email: vitor.roque@ipg.pt

Rui P. Lopes

Instituto Politécnico de Bragança, ESTiG, Bragança – Portugal

email: rlopes@ipb.pt

José Luís Oliveira

Universidade de Aveiro, DET, Aveiro – Portugal

email: jlo@det.ua.pt

Keywords: Policy based network management, policies, network management, transactions.

Abstract

As networks increase in size, heterogeneity, complexity and pervasiveness, effective management of such networks becomes more important and increasingly difficult. In this context, PBNM (Policy-Based Network Management) has been gaining popularity in the recent years. New demands on internetworking, services specification, QoS and generically on network management functionality have been driving users to consider this paradigm in their own networks.

As people start exploiting PBNM, another aspect comes to attention: transactional integrity. Transactional control envisages achieving consistent state changes along the network. In other words, state transition in network devices is only authorized if all the related operations are successfully taken.

In this paper we propose a transactional control mechanism for PBNM systems, namely its assurance across different systems and different network domains.

1 INTRODUCTION

The telecommunications market has suffered an impressive growth during the last years. The quantity of network devices, users and organizations that are interconnected show how the complexity of these networks has increased. The number and diversity of applications requiring network services (virtual private networks, QoS and so on) have also increased. The bandwidth required by many of these applications, such as VoIP or multimedia streaming, is considerably higher from what we have a couple of years ago. Likewise, users' expectations are pointed to crescent demands and to higher service levels. In this context, organizations will need expedited methodologies to configure and manage networks, systems and resources. Policy-based network management (PBNM), one of such methods, has been gaining importance as network dimension and complexity has growing.

The goal behind PBNM is to approximate the business personal from the network technicians, by allowing managing network from high level policy rules. Each policy can represent an action that is applied to thousands of network elements. However, to be effective, this operation should be automated with appropriate software tools and protocols, which allows the policy to be applied regardless of the equipment vendor.

Another challenge in PBNM is how to achieve full transactional integrity in a network domain. Today solutions, namely the COPS or SNMP protocols, permits to achieve transactional integrity at the device level, but when we go to the network level, i.e., when we want to apply the same configuration to all the devices at one network domain we will face some difficulties.

This paper presents a transaction manager and pairs of PDP/PEP which can deal with the properties of a network configuration transaction.

2 BACKGROUND ON POLICY BASED MANAGEMENT

Policies are plans of an organization to achieve its objectives. A policy is a persistent specification of an objective to be achieved or of a set of actions to be performed in the future or as an on-going regular activity.

Policy based networking is the application of these organizational policies in the context of networking [1]. It is usually concerned with the implementation of organizational objectives as automated operations, management and control systems. In this context a policy is a relationship between network objects, such as particular groups of network elements, network resources and services, and user groups. For example, a bandwidth management policy may apply to all routers within a particular region or of a particular type. An authorization policy may specify that all members of a department have access to a particular service [2].

Comparing to conventional network management models, closer to low level instrumentation procedures [3], PBNM simplifies interfaces by extracting commonality across devices; moreover, it provides consistency across interfaces. Network behaviors and management data will be standardized and abstracted. Network actions or

configurations will be derived from these "policy rules", and the policy rules can be differently applied from vendor to vendor. These abstracted management data and fewer interfaces are the keys to achieve better scalability and simplicity in large network.

To support the high level dictated by PBNM, the IETF has proposed a policy framework architecture [2] that we describe in the Figure 1.

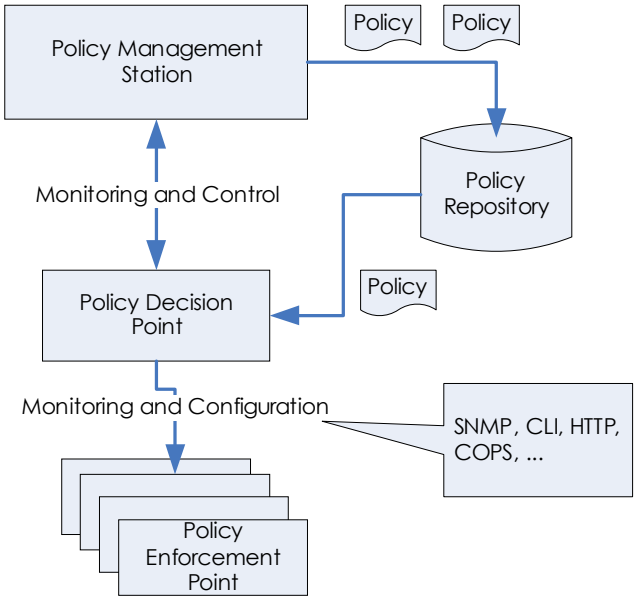

Figure 1 – IETF PBNM architecture.

This architecture describes the key components: Policy Management Tool (Policy Console), Policy Decision Point (PDP) or Police Server, Policy Enforcement Points (PEP) and Policy Repository. A network management protocol, like COPS [4], SNMP [5] or other, is used to transfer management information among network management entities such as agents, managers, decision, and enforcement points. These protocols guarantees transport of information, however network level transactions are still an open issue [6].

A PDP may have a different number of PEPs under its responsibility (Figure 2).

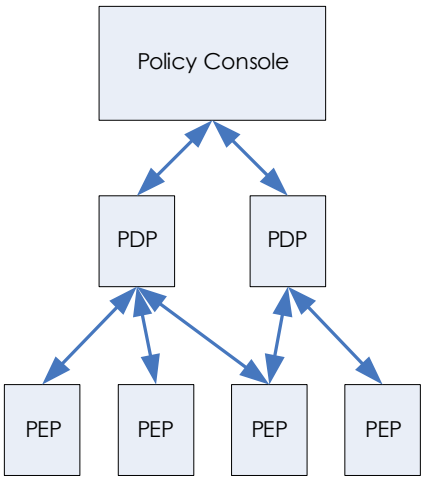

Figure 2 – PBNM architecture hierarchy.

If a single PDP is used to configure a large number of PEPs, we may have some scalability problems (centralized model). If we use more PDPs, we tend to distribute the configuration operations among a more meaningful number of PDPs. If each PDP is responsible for a single PEP, we are under a strong distribution scenario [7].

3 BACKGROUND ON TRANSACTIONS

The concept of transactions has been widely supported by a variety of existing systems, including data-oriented systems, such as databases, and process-oriented systems such as distributed systems. This concept has the same purpose in all these systems and is meant to "group" a set of operations, mainly read and write operations, into one logical execution unit called a *transaction*. Transactions guarantee that the data will be consistent at the end of its execution, regardless of whether the transaction was successful (commits) or have failed (aborts).

Transactions must follow the ACID properties [8]. These properties, identified by *Atomicity*, *Consistency*, *Isolation*, and *Durability ensure that a transaction is performed in a correct way and that it leaves the system in a stable state*.

The atomicity property ensures that all operations performed as a part of a transaction are considered as atomic – i.e. all the transaction operations are performed or none of them. If a transaction aborts, all the operations are undone and the state will roll back to the previous stable state.

Consistency ensures that state changes occur from one consistent state to another. Consistency of a state is defined by a set of constraints and variants which must be satisfied. The property of consistency enables an application to perform a set of operations guaranteed to create a new state satisfying these constraints.

The isolation property is used in situations where multiple processing entities reference and change the same underlying resources and data. An executing transaction cannot reveal its results to other concurrent transactions before it commits.

Durability guarantees that the result of a transaction is durable (persistent) and will not easily be lost (except in the case of catastrophes, such as destruction of the disk and all its backups). Durability is usually implemented by using a persistent storage mechanism.

Taking these concepts to PBNM, the configuration of equipment following a policy usually results in the update of several managed objects and, to be successful, all of them must be configured or none at all. Thus, PBNM must rely on a transaction service so that the network travels between stable states.

4 DISTRIBUTION OF POLICIES

In modern communication networks, an efficient and effective network management system must address support for management of network domains. In this context, the term domain represents a set of interconnected networking devices which may share a common goal or goals – the network wide policy.

Network wide policies imply that a single sentence of objectives be translated into a set of network equipment rules which are then used for equipment configuration

purposes. Each rule should be installed in a specific network component thus contributing to the desired global goal.

Looking from a different angle, each network wide policy represents a transaction, which means that several configuration requests must be clustered so that they are performed with ACID properties. In other words, the requests must be all successfully applied – thus achieving a consistent state transition – or none at all, thus rolling back to the previous consistent state.

In the following examples, and to illustrated the transactions mechanism, we will use the work coming from the several IETF working groups, namely Policy [9], Diffserv [10], and SNMPconf [11].

4.1 Network wide policies as transactions

Let us consider the example of a network composed of several core routers, identified by NE C_* and two edge routers, NE E_1 and NE E_2 (Figure 3). Let us also consider that all the nodes support DiffServ classification of traffic [12] [13].

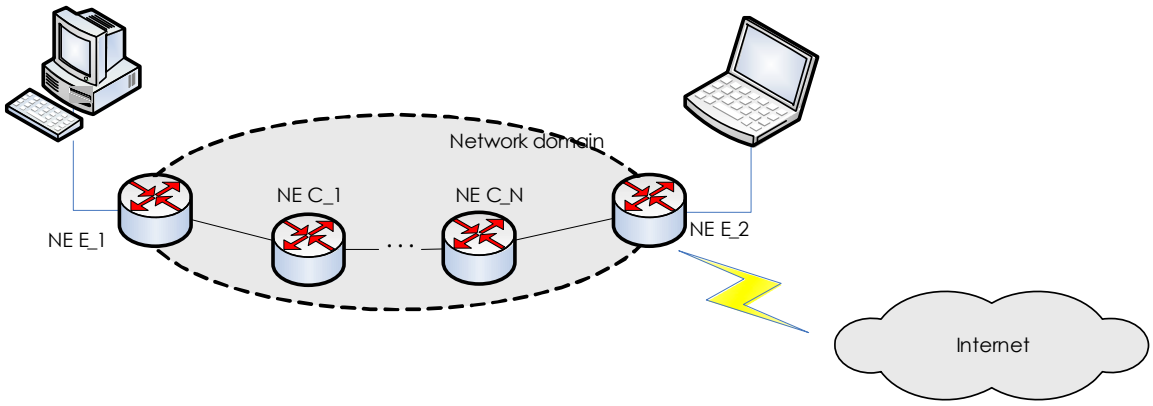

Figure 3 – Network domain policies.

Considering that it is necessary to establish a connection with a given class of service (the network wide policy) we have to configure all the routers along the path and we must guarantee that the communication requirements are met. If, at any point, this is not possible, we have to rollback to the previous configuration.

For simplicity, we consider that the DiffServ nodes are configured through SNMP and/or COPS with the DiffServ MIB [14] and/or DiffServ PIB [15].

The configuration activity taken when applying a policy must be performed as a transaction, thus having the ACID properties. Transactions must be isolated from each other. This property requires some sort of concurrency control to prevent other policies, which might collide with the one we are trying to enforce, from being applied (Figure 4).

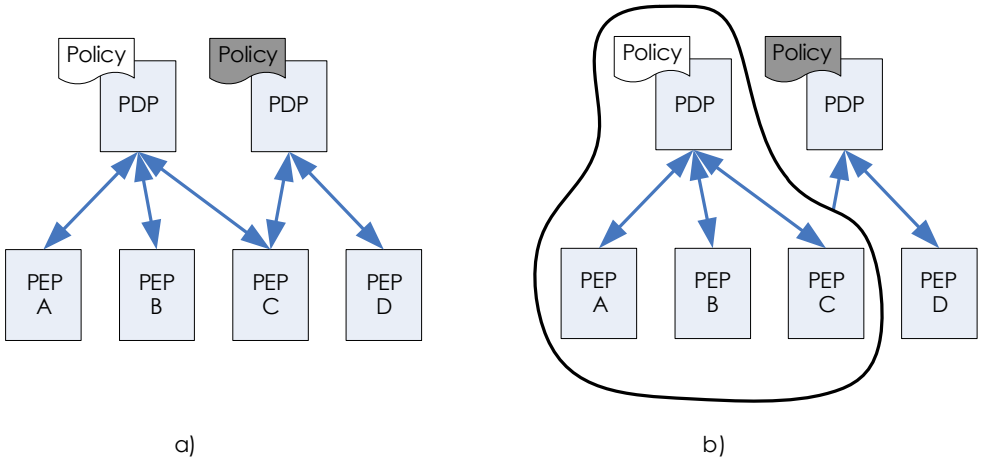

Figure 4 – Concurrency control.
a) Conflicting configuration in PEP C; b) Isolation of transactions.

From a technical perspective, we have to get a write lock to each router under state change. It is well known, from the distributed systems theory that a value can be read simultaneously by several clients. However, modifying a value usually requires dedicated access by each client [16]. This fact alerts us to the possibility of deadlocks as well as starvation.

The SNMP framework defines a mechanism to deal with multiple managers. The standard textual convention document defines *TestAndIncr*, a spinlock, which is used to avoid race conditions [17]. Objects of this type must be set to their current value otherwise the set operation will fail. If successful, its value is incremented.

In a transaction scenario, the manager would have to retrieve the spinlock value from all the PEPs and then try to set a new state with the retrieved value. Should any operation fail, the transaction would have to bring all the PEPs to the previous state. This scenario does not provide an exclusive access lock to the PEP but it provides a method to detect if a different manager tried to configure it.

A different perspective would be to use a transaction manager to grant or deny access to the PDP but this would require that all the requests be sent to it. Current management applications are not built to perform in this way.

The durability property is ensured by the storage resources on network elements. Usually, it relies on flash RAM or on hard drives. This is usually defined through an object of the *StorageType* type [17].

A transaction could only end successfully if the state is consistent. The PEPs must only accept data which drives them to a possible state. In the *DiffServ MIB* [18], several conceptual tables are used for configuration. A conceptual table provides a column, of the *RowStatus* type, which reflects the status of the data stored in the associated row. If the status is 'notReady' then there are missing or incorrect information in it. Only if it is 'active' or 'notInService' the state is, or can, be consistent.

The last property is atomicity. A transaction must be executed completely or not at all. Since we can have policies being applied to different PEPs and that each PEP has no knowledge about the others, the atomicity must be ensured at a higher level – the PDP.

Transaction control is independent of the client and server objects and the operations between them. The coordination role is usually taken by a specific process – the transaction coordinator. The transaction is transparent to the client: it just requests the beginning of a transaction by sending a begin message to the coordinator followed by the configuration messages to the server objects. Finally, the client issues a commit message to the coordinator which will be responsible for ending the transaction.

A well known protocol for achieving atomicity in transactions is the Two-Phase Commit Protocol (2PC) [19]. The 2PC has two phases:

The voting phase: a coordinator process is started (usually at the site where the transaction is initialized), writes a begin commit record in its log, sends a vote message to the participants, and enters the wait state. This message also contains a unique transaction id, which will be used in further messages.

When a participant receives a vote message, it checks if it can commit the transaction. If it can, the participant writes a ready record in its log, sends a vote confirmation message to the coordinator, and enters the ready state. Otherwise, the participants decide to unilaterally abort the transaction by sending an abort message to the coordinator. It enters the abort state and can forget about the transaction.

The commit phase: After the coordinator has received votes from all participants it decides whether to commit or abort according to the global commit rule, and writes this decision in the log. If the decision is to commit, it sends a commit message to all participants. Otherwise, it sends an abort message to all sites that voted to commit. Finally, it writes an end of transaction record in its log.

We thus have an exchange of messages as shown in Figure 5.

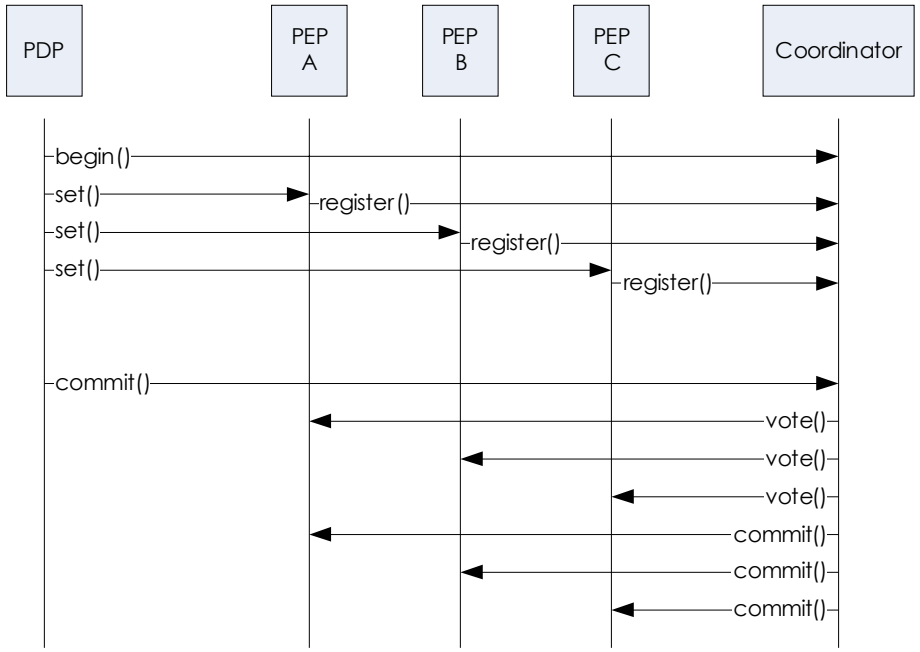

Figure 5 – Messages exchanged for the 2PC.

To cope with eventual crashes or lost messages, the participants will have to provide some recover procedures:

- If the client fails before committing, the coordinator will eventually abort the transaction by sending abort messages to the transaction servers.
- If a server fails before voting, the coordinator should interpret the missing vote as a vote against and should explicitly abort the transaction.
- If the coordinator fails during the voting phase, the servers will not receive the commit message and eventually abort.
- If the server fails after voting and if it has voted in favour, after restarting it will ask the coordinator for the decision taken and act accordingly. If the decision is to commit, it must use the data recovered from temporary storage.
- If the coordinator fails after the first commit message, it has to retransmit the commit request, when getting back online.

Next section will describe how the roles can be applied to policy based management.

5 AN ARCHITECTURE FOR POLICY TRANSACTIONS

This section describes our architecture for implementing transactions in PBNM (Figure 6). Ignoring the Policy Console and the Policy Repository, it closely resembles the IETF architecture, basing the main operations on PDPs and PEPs.

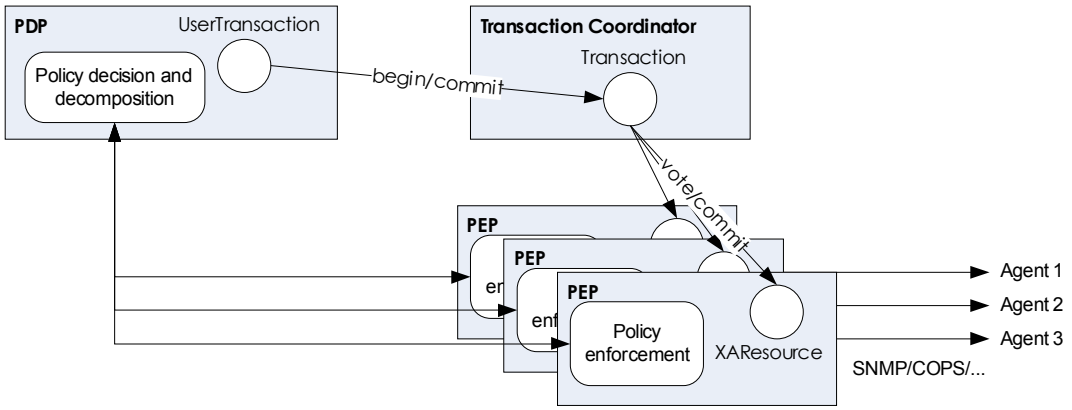

Figure 6 – Architecture for PBNM transactions.

We use the Java Transaction API (JTA) [20] to provide coordination between the PDP and PEPs. The role of the *Transaction Coordinator* is to control and coordinate the transaction, as referred in the previous section. The JTA provides a generic coordinator as part of the API.

The PDP works as a transactions client and the PEPs as servers. In JTA, this means that the PDP has to obtain a reference to a *UserTransaction* object, which will be used to start and end the transactions:

```
public void applyPolicy(Policy policy) {

  UserTransaction ut = context.getUserTransaction();

  try {
    // Request the transaction coordinator to start a transaction
    ut.begin();
    setPolicy(policy);
    // Request the transaction coordinator to commit the transaction
    ut.commit();
  } catch (Exception ex) {
    try {
      ut.rollback();
    } catch (SystemException syex) {
      throw new EJBException("Rollback failed: " + syex.getMessage());
    }
    throw new EJBException("Transaction failed: " + ex.getMessage());
  }
}
```

The PEP is the transaction server object. As such, it has to implement the XAResource interface. This interface provides methods corresponding to the 2PC protocol, namely the voting phase (`prepare`), `commit` and others.

When the PDP send a configuration message to the PEP, it has to register on the *Transaction Coordinator* through the `enlistResource` method of the Transaction interface. This is how the coordinator gets the references to the participants in the transaction. The PEP may have to store some information retrieved from the agent to cope with crashes the may happen.

The implementation of the `prepare` method must also include code to check if the agent is in a consistent state, by querying the conceptual table *RowStatus* column, for example, and if the transaction could be committed. Meanwhile, it also has to check the spinlock object, to see if some other manager is configuring or has configured the agent. In this case, the PEP will vote against committing and the transaction has to be aborted.

After the PDP has issued the commit message, the *transaction coordinator* will request the PEPs to install the policy in the agent.

The PEP also plays an important role in the recovery process in case some message is lost or some participant crashes:

- If the PDP fails before committing, the coordinator will eventually abort the transaction by sending abort messages to the PEPs.
- If a PEP fails before voting, the coordinator should interpret the missing vote as a vote against and should explicitly abort the transaction.
- If the agent fails before voting, the PEP will loose contact with it and will vote against the commit.
- If the coordinator fails during the voting phase, the PEPs will not receive the commit message and, after a timeout, will abort the transaction.

- If the PEP fails after voting and if it has voted in favour, after restarting it will ask the coordinator for the decision taken and act accordingly. If the decision is to commit, it must use the data recovered from temporary storage.
- If the agent fails after the PEP has voted and if the decision is to commit, the PEP will wait for the agent to come back online and restore the state previously retrieved from it. The PEP will then complete the transaction by applying the transaction decision.
- If the coordinator fails after the first commit message, it has to retransmit the commit request, when getting back online.

6 CONCLUSIONS

Policy-Based Network Management is a methodology wherein configuration information is derived from rules and network-wide objectives, and is distributed to many potentially network elements with the goal of achieving a consistent network behaviour.

The configuration activity causes state changes in the network elements and to achieve consistent state changes in network domains it is necessary to guarantee support for transactional integrity.

Within this paper we made some considerations about the importance of to guarantee transactional integrity at the network level in PBNM. The presented architecture, based on the JTA package and on the 2PC protocol, validates our ideas.

7 REFERENCES

1. Chadha, R., G. Lapiotis, and S. Wright, *Guest Editorial - Policy-Based Networking*, in *IEEE Network*. 2002, IEEE.
2. Yavatkar, R., D. Pendarakis, and R. Guerin, *A Framework for Policy-based Admission Control - RFC2753*. 2000, The Internet Engineering Task Force (IETF).
3. Wong, K. and E. Law. *ABB: active bandwidth broker*. in *SPIE ITCom 2001*. 2001. Denver, USA.
4. Durham, D., et al., *The COPS (Common Open Policy Service) Protocol - RFC2748*. 2000, The Internet Engineering Task Force (IETF).
5. Case, J., et al., *Introduction to Version 3 of the Internet-standard Network Management Framework - RFC 2570*. 1999, The Internet Engineering Task Force (IETF).
6. MacFaden, M., et al., *Configuring Networks and Devices With SNMP - RFC3512*. 2003, The Internet Engineering Task Force (IETF).
7. Martinez, P., et al. *Using the Script MIB for Policy-based Configuration Management*. in *IEEE/IFIP Network Operations and Management Symposium 2002*. 2002. Florence.
8. Gray, J. and A. Reuter, *Transaction Processing: Concepts and Techniques*. 1994: Morgan Kaufmann.
9. *Policy Framework (policy) WG*. 2004, The Internet Engineering Task Force (IETF).

10. *Differentiated Services (diffserv) WG*. 2003, The Internet Engineering Task Force (IETF).
11. *Configuration Management with SNMP (snmpconf) WG*. 2003, The Internet Engineering Task Force (IETF).
12. Blake, S., et al., *An Architecture for Differentiated Services - RFC2475*. 1998, The Internet Engineering Task Force (IETF).
13. Grossman, D., *New Terminology and Clarifications for Diffserv - RFC3260*. 2002, The Internet Engineering Task Force (IETF).
14. Baker, F., K. Chan, and A. Smith, *Management Information Base for the Differentiated Services Architecture - RFC3289*. 2002, The Internet Engineering Task Force (IETF).
15. Chan, K., et al., *Differentiated Services Quality of Service Policy Information Base - RFC3317*. 2003, The Internet Engineering Task Force (IETF).
16. Emmerich, W., *Engineering Distributed Objects*. 2000: Wiley.
17. McCloghrie, K., et al., *Textual Conventions for SMIv2 - RFC2579*. 1999, The Internet Engineering Task Force (IETF).
18. Hazewinkel, H. and D. Partain, *The Differentiated Services Configuration MIB - RFC3747*. 2004, The Internet Engineering Task Force (IETF).
19. Özsu, M. and P. Valduriez, *Principles of Distributed Database Systems*. 2nd. ed ed. 1999: Prentice Hall.
20. *The Java Transaction API (JTA)*. 2004, Sun Microsystems.