Transaction-based Configuration Management for Mobile Networks

Henning Sanneck, Christoph Schmelz

Siemens AG, Communications – Mobile Networks
D-81359 Munich, Germany

Alan Southall, Joachim Sokol, Christian Kleegrewe, Christoph Gerdes

Siemens AG, Corporate Technology
Information & Communication
D-81739 Munich, Germany

***Abstract*— Autonomic computing & communication includes the vision of "self-configuring" systems which can adapt themselves to their environment. "Adaptation" means essentially to establish a working initial configuration and to maintain it over time in accordance with the configuration of other systems that have dependencies. In mobile networks, this is a complex task due to the degree of distribution of the network elements (NEs), the properties of the Operation, Administration and Maintenance (OAM) network, the specific configuration dependencies between the network elements and the hierarchical nature of the legacy management infrastructure.**

To cope with these characteristics, an approach is presented, where configuration changes are communicated as transactions between the NEs and their element management system (EMS). A master-replica paradigm is adopted where the replicas (=NEs) are allowed to autonomously commit configuration data as "tentative" and the master (EMS) follows the state of the replicas, but may force rollbacks on them.

Configuration dependencies between different NEs are expressed as transaction groups. The execution of these transaction groups is controlled by a centralized transaction manager located at the EMS. To provide for a defined duration of the transaction group execution in the presence of transaction failures, either a complete rollback or a partial commit of the transaction group (thus re-assessing the configuration dependencies) is possible based on operator policies.

The presented scheme allows improving the level of configuration data consistency and the degree of automation in the configuration operations with benefits for both the manufacturer and the mobile network operator. A proposed system architecture and details of an experimental implementation are presented.

A. INTRODUCTION

Today, most network management systems are still based on a hierarchical infrastructure (e.g., mobile networks according to 2G/3G standards, fixed networks based on ITU and IETF standards). The configuration work cycle in such a system is characterized by the rollout of configuration data from a central Element Management System (EMS) to numerous Network Elements (NEs) and the alignment of the actual configurations of the NEs towards the EMS. The described work cycle is triggered by updates of the network planning or changes resulting from the day-to-day operation.

The configuration tasks in OAM ranging from initial installation and commissioning of NEs up to reconfiguration of network domains in operation still require significant manual interventions by a human operator. To reduce the amount of manual work and the associated significant costs, the vision of autonomic computing / communication [1] seems appealing.

Figure 1 shows how advances in configuration management automation could be mapped to the different levels in autonomic computing / communication (note that the next level always includes the functions of the previous one). A crucial boundary is between the levels of "Predictive" and "Adaptive" where the scope of automation is extended to NE groups rather than just individual NEs.

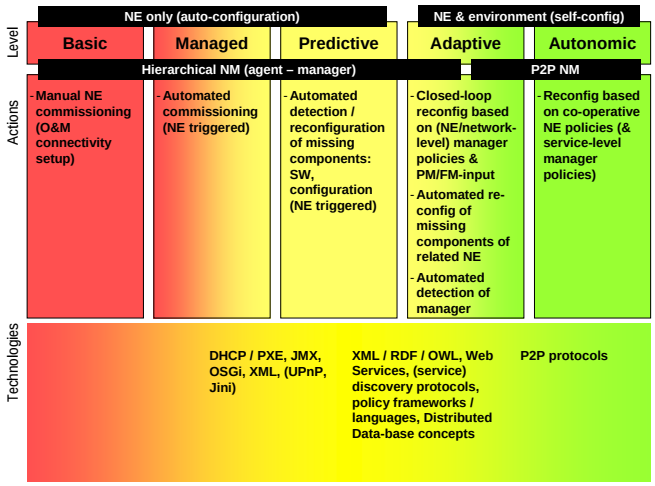

Figure 1 Steps in Network Configuration Management Automation

With regard to the characteristics of a mobile network ([2], [3]) as a potentially "autonomic" system, it can be observed that the majority of the individual sub-systems (i.e., network elements like radio base stations) are less complex than in the IT domain. However, the degree of distribution and the wide-area network environment

results in more challenging issues of scale. The relevant characteristics can be summarized as follows:

- Number: Thousands of NEs (base stations / access points) may be associated with one single EMS.
- Reliability: NEs (particularly in rural areas) are attached over relatively expensive and thus bandwidth-limited links. Furthermore the connectivity on those links can be unreliable due to the employed network technology (microwave radio links) and the priority of user traffic over O&M traffic. Yet, if the O&M connectivity is not present for a certain period of time, the NE should still function correctly in the operational network.
- Duration: Configurations may change within short timescales at the NE and EMS.
- Concurrency: Multiple sources of configuration changes may be active concurrently (network planning, several human operators doing day-to-day configuration changes, local changes by the field service)
- Service availability: Users expect a high availability of services, i.e. service-affecting configuration changes can only be rolled out during defined short time windows (night hours, weekends). It may thus happen that parts of a network configuration change may need to be cancelled / rolled back when a time limit is reached.

These characteristics require a configuration management system that is able to detect and automatically react to configuration data consistency errors between an NE and its EMS as well as for groups of NEs that have dependencies. Reliability should be provided by means of rollbacks to provide consistency even after logical or consistency errors that cannot be easily solved.

The accepted means to realize the described functionality are transactions-based systems ([4]), which are defined on the lowest level by the ACID (Atomicity, Consistency, Isolation and Durability) paradigm. We extend this paradigm towards "distributed adaptive transactions" to provide for scalable synchronization of configuration data across NE groups and the EMS. Based on our initial work on the topic [5] this paper gives an overview of the distributed adaptive transaction concept and the transaction generation process. Then, further results on transaction execution, integration into an EMS, an experimental implementation and its evaluation are presented.

B. DISTRIBUTED ADAPTIVE TRANSACTIONS

The binary "accept" or "reject" of delta and bulk configuration changes made to a group of NEs in current element management systems follows the strict ACID paradigm on the fine granularity of an atomic operation, resulting in a very coarse view of the dependencies between the changes on the individual NEs. Thus, when errors have occurred, significant expensive manual intervention of the human operator is required to enforce a rollback across the entire group of NEs, to detect the sources of the errors, and to finally reach a consistent network state. The more likely the following attributes apply, the more likely an atomic operation (two-phase-commit / "all or nothing" strategy) will not be feasible anymore:

- High probability that a specific NE is not reachable at a given point in time
- Large number of NE groups with dependencies
- Large total number of NEs having dependencies
- Large number of different managed objects with dependencies

Therefore, to accommodate the distribution characteristic, a transaction hierarchy is introduced:

- NE transactions (NET): An NE transaction transforms the configuration data of one NE from a consistent state to another consistent state. Note that a single NET may consist of multiple atomic operations (e.g., insert, update, delete operations).
- Transaction groups (TG): A transaction group contains a set of NETs that have inter-dependencies. Once the TG is committed, the group of NEs is transferred from one consistent state into another consistent state. TGs with no interdependencies may be executed in parallel.
- Network transaction (NT): A network transaction consists of one or more TGs applicable for an arbitrary number of NEs. NTs are independent of each other and are executed consecutively.

Furthermore, it is desirable to have an "adaptive" approach to provide for a flexible control over the transaction execution process and long lasting execution periods that can continue even up to days. The flexibility refers to re-assessing (and possibly revoking some of) the dependencies based on the result of individual transactions. This means that instead of performing a complete rollback, the system should allow rescheduling of transactions. For example, if a network operator wishes to update *n* NEs with dependencies in a

distributed transaction, but only one single NE is offline during the update process, it is obvious that the distributed transaction shall not fail as a whole. Instead, the operator can often accept a partial success in that *n-1* elements can commit their updates and the remaining single update will be rescheduled to a later point in time where the characteristics described above (reliability, concurrency) may have changed.

To demonstrate the differences between the current and the envisaged workflow, Figure 2 shows the example of cell adjacency management, where a handover (HO) parameter set for a certain cell and the cell adjacency information (ADJC) of neighboring cells pointing to that cell need to be concurrently updated.

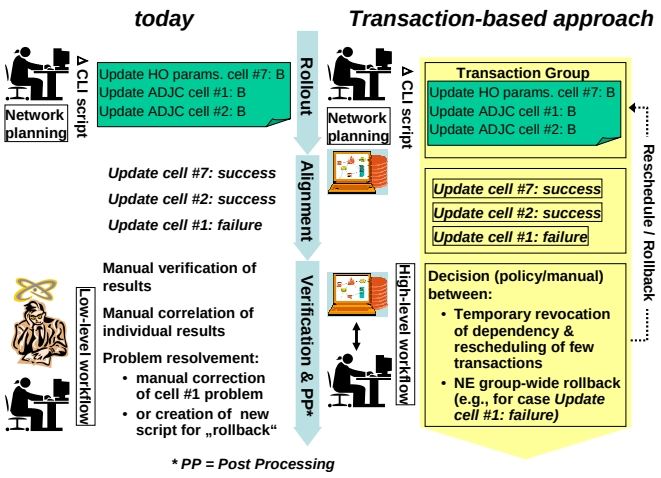

Figure 2 Workflow comparison

Configuration Work Cycle

Based on our approach the modified configuration cycle (cf. Figure 3) consists of the phases of *network planning*, *transaction generation* and *transaction execution*. As soon as a network update is planned, appropriate transactions are generated and their execution is triggered. The result of the generation phase is an NT together with parallelization and execution optimizations. At the beginning of the execution phase the parallelization plan is utilized to schedule individual TGs. The execution plans for each TG fix the sequence of the NET execution.

In the *transaction generation phase* first a dataflow analysis is conducted. Thereby it is analyzed what and when data is written or read. The result of this step is a graph-like representation where the program statements constitute the nodes, and data relationships constitute the edges.

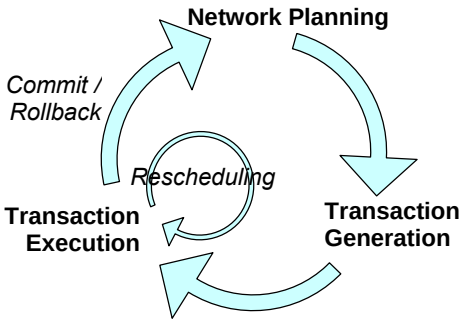

Figure 3 Configuration Work Cycle

In a *parallelization* step this graph is partitioned such that each partition represents an independent set of instructions that can thus be executed in parallel (“transaction line”, TL). Depending on the semantic tree and hence implicitly on the type of the high level input the partitioning of the graph can become highly complex. With regard to the described mobile network environment, a restriction on recursive free script languages seems appropriate.
Even with knowledge of the data model there might be dependencies that remain difficult to detect. One such example is parameter reuse where values are assigned from a finite value set and two parameters must not have the same value. To detect these kinds of dependencies the parallelizer can consult a policy database. Policies within this database have knowledge of specific and arbitrary complex dependencies as well as data models and offer appropriate parallelization strategies.

Before the actual execution can begin an *execution planning* step is conducted. The purpose of this optimization step is to reduce the rollback complexity in case of failing transactions. Generally the generated NETs may have differing degrees of dependency. For example the modification of a cell’s handover information (HO in Figure 2) requires to update the adjacency information of all referring cells. Therefore the degree of dependency is high. On the other hand, a transaction that adjusts a single adjacency information (ADJC) has a low degree of dependency. Transactions with high dependency degrees are typically more critical in their execution, i.e., they are more likely to fail. Instead of treating each NET equally and initiating their execution in parallel, an execution plan is generated. Thereby the planner evaluates the different dependencies and assigns weights which quantify a dependency’s impact on the execution robustness. Since the nature of dependencies varies with the respective NEs,

dependency weights are not globally defined. Rather the creation of the plan is controlled through various policies. Thereby a policy determines the dependency weights and the resulting plan structure.

The *transaction execution* is the most complex phase in the cycle. In the event of a failure the execution process itself becomes a cyclic process where new groups are generated by the re-scheduler and executed until all NETs are successfully committed (or passed to the operator and/or being subject for system policies). The result of the execution phase is a set of successfully executed and committed NETs (e.g. a partially updated network) and, if any, a set of NETs that did not yet commit and can be passed back to the operator or planning application. The transaction execution is detailed in the next chapter.

C. Data Management Model and Transaction Execution

Figure 4 shows the generic data management model, which can be easily mapped to existing systems. The data management model and architecture base on the X/Open DTP reference model ([6]) for transaction systems. A master-replica role mapping is chosen where the EMS is the master and the NEs are replicas with an appropriate degree of autonomous actions based on a "tentative commit" of configuration data.

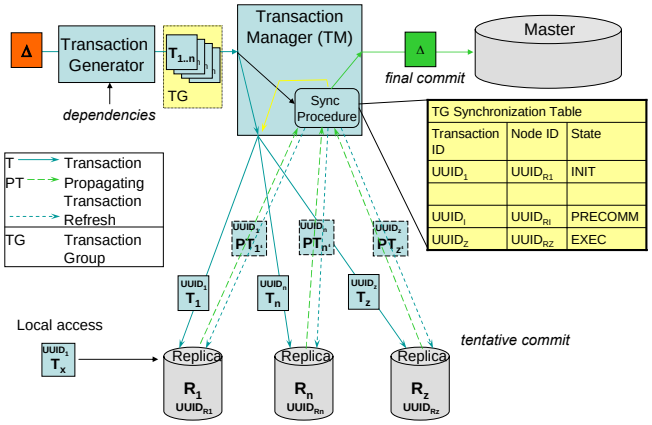

Figure 4 Generic data management model

The work sequence for configuration roll-out / alignment depicted in Figure 4 can be summarized as follows:

- Transactions (T) of one TG register with a "synchronization procedure"
- Transactions are sent to the replicas
- Execution / tentative commit at replicas
- PT (Propagating Transactions): Changes made in the replicas must be propagated to the master to be "finally" committed. These "propagating" transactions are created in the replica and are sent to the master (via TM) after they have been tentatively committed in the replica.
- Depending on the success / failure of the entire TG, changes are finally committed at the master, or the replicas are rolled back

Modifications conducted either through the EMS or through a local terminal at the NE are introduced at the replicas only, as shown in Figure 4. Then, replicas propagate the changes to the master database in the EMS. This way of operation has been chosen because data in the replica may be changed by another source than the master, and such changes should always be committed first (though only "tentatively") in the replica until they are "finally" committed by the master. The EMS is therefore responsible for any consistency checks across the NEs and any decisions as to which updates are to be finally committed, rolled back or postponed to a later time. Thus, the NEs remain the masters of their data, i.e., modifications made can be immediately activated at the NE, but the EMS has the final decision to roll-back or keep the updates.

The Transaction Manager (TM) at the master schedules the network configuration update process. It triggers its execution units (threads) to fetch a TG from their respective transaction line (TL) and executes its NETs on the system according to the execution plan. In the event of a failure during the execution of an individual NET it must be decided whether the whole TG and respectively the whole NT is rolled back, or a rescheduling process is initiated that allows the problematic NET to be isolated and rescheduled for execution at a later point in time. The decision to roll back or partially commit and reschedule is left to an operator or derived from a policy based decision system. If it is decided not to rollback the whole NT but partially commit it, the TG holding the failed transaction T_{error} must be analyzed to find dependent transactions like

$$TG_{rollback} = \{\forall T \in TG; depends(T, T_{error})\}$$

The rest of the group

$$TG_{committed} = TG \setminus TG_{rollback}$$

will be committed. The TM spawns a new execution unit and schedules $TG_{rollback}$ onto its TL for postponed execution. This isolation and rescheduling process is executed on every TG that is scheduled in the same TL as $TG_{rollback}$ (e.g. all TGs that include NETs that depend on T_{error}). Surely the isolation process which may include

the temporary revocation of dependencies must comply with all consistency claims and therefore requires policy support. From an abstract point of view the revocation of a dependency is backed by spawning an execution branch that develops independently from the parallel execution. If the branch leads to success it can be merged with the global view, otherwise the branch is discarded and the global view remains unaffected. To demonstrate this procedure a simple example is considered (see Figure 5): Given is a network update that contains just two transaction groups (TG_1 and TG_2). Amongst others, TG_1 contains a NET (NET_1) that resets a parameter from a finite value set *S*. Further it is assumed that at any time parameters of arbitrary elements must not have the same value, i.e. there exists an implicit dependency between all NETs modifying parameters with value of the set.

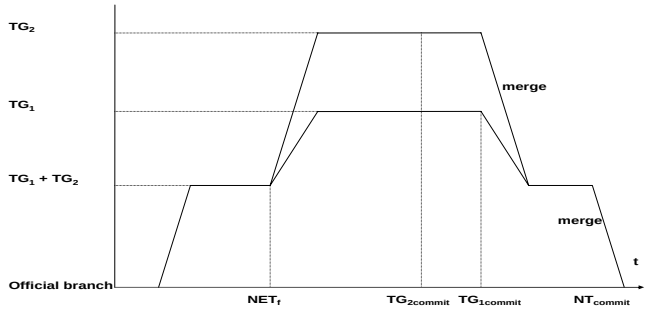

Figure 5 Branching in the presence of failed transactions

As it turns out TG_2 has a similar NET (NET_2) that aims to set a parameter of another element to exactly the same value as the element to be modified by NET_1 is set to at the present time. During execution of TG_1 one NET (NET_f) fails. It happens that NET_f and NET_1 have a strong dependency that cannot be revoked. For the sake of simplicity it is assumed that the implicit dependency between NET_1 and NET_2 is the only dependency that exists between TG_1 and TG_2. Then a rescheduling policy might propose to revoke that dependency thus causing a separate execute branch to spawn.

The branch contains a modified global state that marks points of conflict blindly, i.e. the parameter in question is an assigned value $v \notin S$. While TG_1 is rescheduled to a separate TL still utilizing the yet consistent global state, TG_2 adopts the newly spawned branch with a conflict free state. When TG_2 commits, the changes become visible and the network state becomes inconsistent. The network remains in this state until TG_1 can be committed. As soon as this happens both execution branches are merged. On the contrary, if the merge of branches fails, the official branch, in this example the branch TG_1, will overwrite the spawned branch and all changes of TG_2 are compensated. TG_2 will then, however, be rescheduled and might then commit successfully in the official branch. In any case consistency is restored either when branches are merged or both groups commit on the official branch.

While the TM provides basic interfaces for execution branching and merging it is the responsibility of the respective rescheduling policy to maintain global execution consistency. A general design concept of the TM is to keep core functionality as lightweight as necessary and provide rather extension points for advanced features.

D. Proof-of-Concept Implementation and Evaluation

To show the capabilities of our novel approach, an implementation was successfully realized based on standard Java technologies. The database connection is provided through a JDBC driver that would allow to employ any JDBC compatible database. In addition to the implementation of the core concepts a demonstrator has been developed that visualizes the internal methods of operation of the transaction processing, and furthermore demonstrates the cell adjacency management use case.

The whole software package realizes all key concepts described in this paper and includes the following key building blocks of:

- transaction manager including support for failure handling and multiple policies
- transaction generator with focus on optimization (parallelization and execution planning)
- high level abstraction towards the database systems
- statistics and monitoring utilities
- console for operator interaction
- visualization of the transaction execution
- visualization of the cell adjacency management use case

The class diagram in Figure 6 shows a small excerpt of the *transactionManager* package. It shows the classes that accomplish the infrastructural support for transaction management.

The singleton *TransactionManager* class constitutes the base implementation of the transaction manager. The class allocates the needed execution units and acts as central event manager. Furthermore, it initializes the execution state by downloading a snapshot from the master database and converting the objects to transactional entities in the transactional state. Transactions are accepted through the *addTransaction()*

method. Although an arbitrary number of transactions can be added to the *TransactionManager* it is always assured that only one top level transaction (NT – *NetworkTransaction*) is executed at a time. In addition to the transaction itself, an execution plan generated by the optimizer can be set. Depending on the *ExecutionPlan ExecutionUnits* are added or removed from the manager, thereby an *ExecutionUnit* object implements directly the concept as described in Section C.

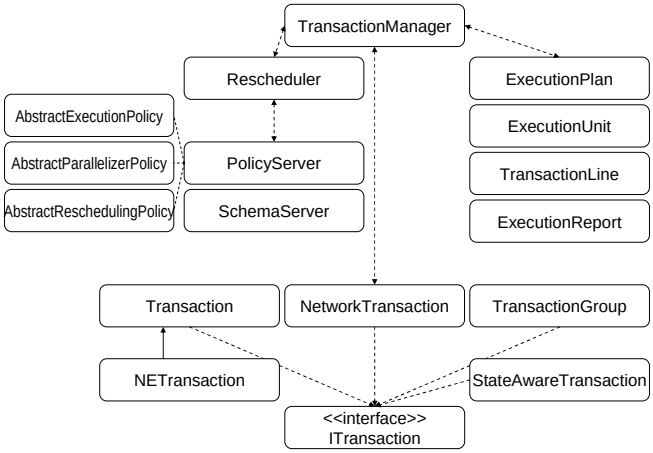

Figure 6 Class overview (excerpt)

To retrieve information on the ongoing execution process, interested objects can register a listener with the *TransactionManager*. Respective events inform of the lifecycle of the different transaction types as well as internal information e.g. execution unit activity and the like. Transaction lifecycle information is passed through the *ExecutionReport* object. It holds information on the transaction, its executing TL, its state (executing, failed, done), and in the case of an error some information on the cause of the error.

In the event of an error the *TransactionManager* requests the *Rescheduler* to extract an applicable policy that analyses the group in question and creates new groups. With the result of the *Rescheduler*, the *TransactionManage*r spawns one or more new *ExecutionUnits* and starts their operation. In order to use resources efficiently *ExecutionUnits* deactivate themselves once they reach an idle state.

To support the development of a database independent transactional state, the execution package provides a transactional and a recoverable object class. The classes feature methods to set and retrieve values with the option to restore a previous state. Once a transaction enters its execution method a separate branch for that transaction is allocated in the associated state instance (e.g. global state or unit state). On commit, it is first tried to merge the transaction branch with the official branch, whereby the official branch is the branch being synchronized with the persistent data store. If the merge completed, the official branch is then pushed to the respective databases via the database abstractions. Especially important are the adapter classes for the different policies: *AbstractExecutionPolicy*, *AbstractParallelizerPolicy* and *AbstractRescheduling Policy*. New policies inherit from these abstract classes and thus can introduce specific domain knowledge for particular use cases.

An extensive evaluation of the implemented system has been done [7] with regard to the following characteristics:

- Scalability to large numbers of replicas
- Resilience to connectivity interruptions
- Performance (network bandwidth consumption and transaction execution delays)
- Parallelization gains

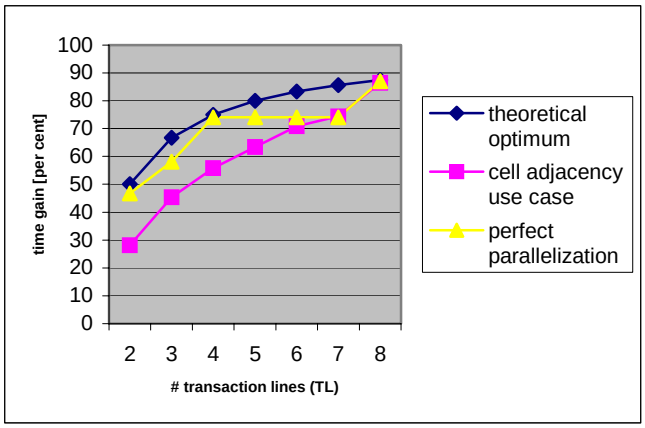

Figure 7 Relative parallelization gain

As an example, Figure 7 shows the relative parallelization gain. The upper curve gives the "theoretical optimum" of $1-\frac{1}{n}$, i.e. the achievable parallelization gain under the assumption of full symmetry of *n* transaction lines. "Perfect parallelization" is the result when driving the experimental system with an NT of eight TGs, which can always be "perfectly" parallelized. "Perfectly" means here, the distribution of the eight TGs over the TLs is as perfect as possible, e.g., for four TLs, two TGs are associated to each line.

Note that when employing five, six or seven TLs the result does not change basically, as at least one of the TLs still needs to accommodate two consecutive TGs. The differences between the "perfect parallelization" and "theoretical optimum" curve for two, four and eight TLs

show the delay impact of the execution of the parallelization process itself in the experimental system.
For the "cell adjacency use case" it can be observed that for only two TLs the parallelization gain is quite far away from the "perfect" case. This is due to the often unequal distribution of TGs to the two TLs (e.g., the actual dependencies may require that six consecutive TGs are associated with TL 1 whereas only two TGs are in TL 2. Thus it is the first TL which causes a 50% delay increase with regard to the perfect case). For an increasing number of available TLs the parallelization gain moves closer towards the optimum, as increasingly often the availability of more TLs can be exploited by the actual distribution of TGs in the "cell adjacency" example.

E. POLICIES

To demonstrate the correct working of the policy mechanisms one parallelization, one execution planning and several rescheduling policies have been implemented in our experimental environment. The *parallelization policy* parallelizes by first checking which transactions of a certain TG work on the same data sets, i.e. the same NE, and accordingly sorts the TGs to TLs. It is then checked if there exist cross dependencies between different NEs, and in case there are, the two TLs will be merged into one. The *execution planning policy* identifies that the TG with the most critical transactions and schedules it to be executed first. The *rescheduling policies* include a policy which defers the execution of a failed transaction until the transaction can finally be committed. In another policy the failed transaction is forwarded to the operator console and the execution is halted until the operator decides on actions to be taken. Other *combined policies* allow expressing full operator logic behind a decision to finish a distributed transaction (like e.g., "commit at the master, if 90% of the individual transactions of the TG have been committed at the replicas"). Thus, the decision process at the EMS can be automated, since the TM as the controller component is autonomously able to control and correlate the status of all individual transactions.

F. INTEGRATION

Figure 8 shows how the described data management subsystem could be integrated into an EMS. The transaction compiler / manager is driven by configuration delta scripts emitted by a configuration preparation tool. The TM interfaces on one side to the master DB at the EMS as well as the human operator and on the other side to the replica DBs at the NE. The TM can employ any transaction-oriented protocol for communicating and transporting state between master and the replicas. Applications can both retrieve configuration state from the master DB ("committed read") as well directly via the legacy configuration management protocols from the NE ("dirty read").

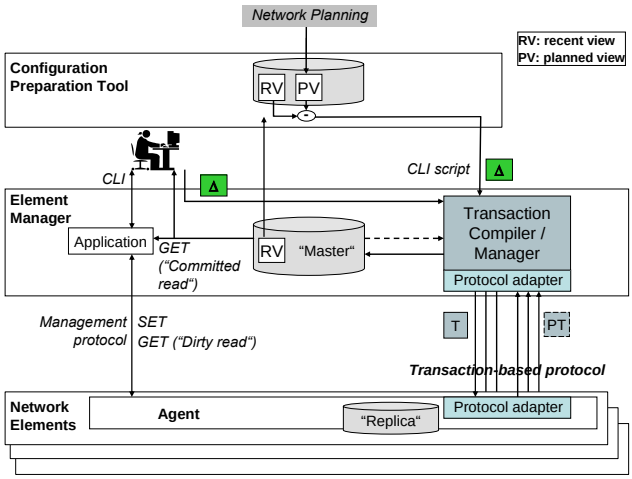

Figure 8 Integration into the element management architecture

G. CONCLUSIONS

A novel concept to enable "distributed adaptive transactions" for configuration management with an arbitrary desired level of control has been presented. Figure 9 shows a mapping of the major building blocks to the MAPE (Monitor-Analyze-Plan-Execute) model of autonomic computing.

By employing transaction groups covering several NEs, dependencies between NEs can be expressed in the "plan" phase and the execution of configuration changes along these dependencies can be controlled ("distributed" transactions). The use of transactional semantics between an individual NE and the EMS assure that the individual NE and the EMS always converge to a consistent state. While the NE can autonomously update and activate configuration data ("actuator", cf. Figure 9), the EMS retains the decision power to rollback any NE configuration. Moreover, while the EMS can rollout new configuration data, it is assured that data at the NE is never overwritten without a notification of the conflict to the NE and the EMS ("sensor" function of the NE).

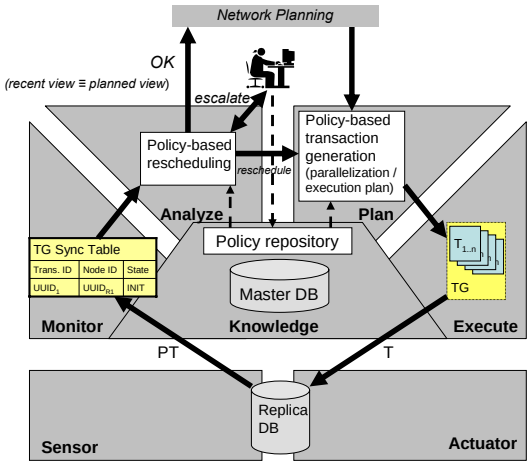

Figure 9 Mapping to the MAPE model

The distributed transaction mechanism can be made "adaptive" in the sense that the original expression of dependencies may be revised during the execution of the distributed transaction. Dependent on the overall state of the distributed transaction ("monitor", cf. Figure 9), it can be rolled back or individual NE transactions can be committed at an operator-defined time to finish the configuration process. This considerably improves an "all-or-nothing" approach (two-phase-commit). Given that a significant probability for connectivity interruptions exists, two-phase-commit becomes unsuitable with increasing size of a transaction group due to the incurred overhead in network traffic and processing for roll-backs.

The capability to control the network configuration on an NE group rather than individual NE level in the described way thus provides the basis to reach a higher degree of automation. On one hand, this directly allows to relieve the human operator of the treatment of simple, recurring problems in the roll-out of network configurations. On the other hand, by correlating the states of transactions ("analyze", cf. Figure 9) more intelligent decisions in the configuration process are possible. Furthermore, it is feasible to include other available state information like alarms into the decision process.

With its policy-based, network-wide configuration management, a higher-level interface to the operator is provided, thus ultimately relieving him of low-level per-NE configuration tasks. At the level of the operator a group-wide rollback thus may appear only as a single operation (dependent on the sophistication and thus the number of necessary escalations to a human operator).

Thus an automation level of "predictive" (cf. Figure 1) with some elements of the "adaptive" level can been reached. However, this can only be achieved by encapsulating today's human operator knowledge into the policies which drive the transaction generator and manager.

However, the process of how these policies are developed and how complex (how intelligent) these policies ultimately can (or should) be is clearly an area of future work. In principle there seems to be any degree possible between complete automation and only manual operator control. We assume policy systems will evolve from atomic policies towards an integrated policy developing system, consisting of an integrated programming environment for developing very complex policies in a high level language. Nevertheless, to solve these issues, clearly operational experience from a large real system implementing the described transaction concept is required.

REFERENCES

[1] A. Ganek, T. Corbi, "The dawning of the autonomic computing era", IBM Systems Journal, Vol. 42(1), 2003.

[2] O. Lazaro, J. Irvine, D. Girma, J. Dunlop, A. Liotta, N. Borselius, C. Mitchell, Management System Requirements for Wireless Systems Beyond 3G, Proceedings - IST Mobile & Wireless Communications Summit 2002, Thessaloniki, Greece, June 2002, pp. 240-244, http://www.isrc.rhul.ac.uk/nb/publications/IST2002.pdf

[3] Wireless World Research Forum (WWRF), Book of Visions 2001, Version 1.0, 2001, http://www.wireless-world-research.org/

[4] J. Gray and A. Reuter, Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993.

[5] H. Sanneck, J. Sokol, C. Gerdes, C. Schmelz, A. Southall, C. Kleegrewe, Mobile network self-configuration based on distributed adaptive transactions, Praxis der Informationsverarbeitung und Kommunikation (PIK), 28(4):217--222, October 2005.

[6] The Open Group www.opengroup.org, X/Open Distributed Transaction Processing : Reference Model version 3, 1996

[7] A. Däubler, Performance evaluation of a Network-Wide Subsystem for Configuration Data Management, Master's thesis, University of Augsburg, October 2005.