TOWARDS ROBUST CONFIGURATION OF ADVANCED PRIVATE LAND MOBILE RADIO SYSTEMS

by

Roland James Chang
B.A.Sc. (Honors) Simon Fraser University 1999

PROJECT SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF ENGINEERING

In the
School
of
Engineering Science

© Roland James Chang 2006

SIMON FRASER UNIVERSITY

Summer 2006

All rights reserved. This work may not be reproduced in whole or in part, by photocopy or other means, without permission of the author.
Approval

Name: Roland James Chang
Degree: Master of Engineering
Title of Thesis: Towards Robust Configuration of Advanced Private Land Mobile Radio Systems

Examining Committee:

Chair: Dr. Atousa HajShirMohammadi
Lecturer of Engineering Science

Dr. John C. Dill
Senior Supervisor
Professor of Engineering Science

Dr. R.H. Stephen Hardy
Supervisor
Professor of Engineering Science

Date Defended: May 15, 2006
DECLARATION OF PARTIAL COPYRIGHT LICENCE

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection, and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author’s written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, BC, Canada
Abstract

Newer, more advanced, Private Land Mobile Radio (PLMR) systems have many features based in software – touting increased flexibility to better meet the requirements of private radio customers. However, these same features tend to rely heavily upon configuration – which should be robustly designed to cope with network upgrades, equipment failures, and user errors. Towards that end, this paper puts forth some practical configuration design guidelines based upon past PLMR project experiences. Anecdotal retrospectives illustrate how the guidelines can facilitate software development, improve overall system robustness, and even help to resolve system-level usability issues.

Keywords:

Software radio, Trunked radio, Mobile communication systems, Wireless communication systems, Human computer interaction
Acknowledgements

I would like to thank Jerry Nieman and Jim Porter for advising, directing, and supporting my software development efforts at M/A-COM Vancouver which provided the basis for this paper. I would also like to thank Dr. John C. Dill and Dr. R.H. Stephen Hardy for agreeing to be my supervisors, and for their guidance, encouragement and patience throughout my graduate studies.

I would also like to thank all my co-workers and associates at M/A-COM Vancouver, Lowell, and Lynchburg that I have had the pleasure to work and interact with over the years, many of whom will find the anecdotal discussions in this paper familiar.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approval</td>
<td>ii</td>
</tr>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>iv</td>
</tr>
<tr>
<td>Contents</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vi</td>
</tr>
<tr>
<td>Glossary</td>
<td>vii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Challenges</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Remote Geographical Access</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Makeup of the Development Organization</td>
<td>4</td>
</tr>
<tr>
<td>2.3 System Procurement, Human Computer Interaction, and Safety Issues</td>
<td>6</td>
</tr>
<tr>
<td>3 Robust Configuration Design Guidelines</td>
<td>8</td>
</tr>
<tr>
<td>3.1 About Database Management Systems</td>
<td>8</td>
</tr>
<tr>
<td>3.2 Some Brief Terminology</td>
<td>9</td>
</tr>
<tr>
<td>3.3 Guideline G1_TransactionStamp</td>
<td>10</td>
</tr>
<tr>
<td>3.4 Guideline G2_ConfigPropagation</td>
<td>13</td>
</tr>
<tr>
<td>3.5 Guideline G3_BatchAndDelta</td>
<td>15</td>
</tr>
<tr>
<td>3.6 Guideline G4_InformationHierarchy</td>
<td>17</td>
</tr>
<tr>
<td>4 Applying the Design Guidelines: Anecdotes and Examples</td>
<td>23</td>
</tr>
<tr>
<td>4.1 Replacing Connectivity-based Synchronization</td>
<td>23</td>
</tr>
<tr>
<td>4.2 Gracefully Applying Large Scale Resynchronizations</td>
<td>25</td>
</tr>
<tr>
<td>4.3 Isolating Configuration Propagation Effects</td>
<td>28</td>
</tr>
<tr>
<td>4.4 A High-Availability System</td>
<td>31</td>
</tr>
<tr>
<td>4.5 Add/Delete as a Shortcut for Change</td>
<td>33</td>
</tr>
<tr>
<td>4.6 Over-Applying Referential Integrity</td>
<td>34</td>
</tr>
<tr>
<td>5 Conclusion</td>
<td>38</td>
</tr>
<tr>
<td>References</td>
<td>40</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1. Diagram of $G1\_TransactionStamp$ .......................................................... 10
Figure 2. Diagram of $G2\_ConfigPropagation$ .......................................................... 14
Figure 3. Diagram of $G3\_BatchAndDelta$ ............................................................... 16
Figure 4. General Illustration of Foreign Keys .......................................................... 18
Figure 5. Database Relationship Diagram Showing Foreign Keys ............................ 19
Figure 6. Non-orthogonal Use of Foreign Keys .......................................................... 21
Figure 7. Connectivity-based Synchronization .......................................................... 23
Figure 8. Over-Extended "Delta Reload" vs. Hybrid Method vs. Proper Batch
          Configuration Processing .................................................................................. 27
Figure 9. Two Configuration Propagation Arrangements ......................................... 29
Figure 10. Applying $G1\_TransactionStamp$ and $G2\_ConfigPropagation$ to a High-
            Availability System ....................................................................................... 32
Figure 11. Referential Integrity Logic Distribution in a Voice Switch Configuration
            Example ............................................................................................................ 35
<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBMS</td>
<td>DataBase Management System</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HCI</td>
<td>Human Computer Interaction</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>LOC</td>
<td>Lines of Code – common measure of software size and/or complexity.</td>
</tr>
<tr>
<td>PLMR</td>
<td>Private Land Mobile Radio</td>
</tr>
<tr>
<td>RFP</td>
<td>Request for Proposal</td>
</tr>
<tr>
<td>RFQ</td>
<td>Request for Quotation</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-Time Operating System</td>
</tr>
<tr>
<td>Talkgroup</td>
<td>Representation of a scarce PLMR system resource that allows voice communications amongst a specific set of radio users. Each talkgroup is usually associated with a well-known, unique identifier. Talkgroups are the fundamental building blocks of PLMR system configuration.</td>
</tr>
<tr>
<td>Voice</td>
<td></td>
</tr>
<tr>
<td>Trunking</td>
<td>“Trunking” is a general term borrowed from telephone systems to convey situations where a relatively small number of communication paths provides voice services for a relatively large number of users. In the context of PLMR systems, voice trunking usually means the sharing of scarce airlink (i.e. frequency channel) resources amongst many users. Newer, digital-trunking algorithms are usually much better at using frequency channel resources efficiently.</td>
</tr>
</tbody>
</table>
1 Introduction

As defined by the Federal Communications Commission (FCC), Private Land Mobile Radio (PLMR) systems are used by companies, governments, public safety agencies, transportation authorities, utilities, and other organizations to meet a wide range of communication needs, including coordination of people and materials, important safety and security needs, and quick response in times of emergency [1]. As PLMR systems advance from conventional analog voice architectures towards connectionless, IP packet-based, digitally-trunked voice and data networks requiring the use of digital signal processing (DSP) technologies – companies in the PLMR industry often experience shifts from hardware development methodologies towards software development methodologies. Often accompanying such shifts – are corresponding progressions towards innovative software-based features capable of shaping and affecting the behaviours of a PLMR system during live operation. Indeed, software-based features are fast becoming the industry norm – and prompting a revolution towards advanced, agile PLMR systems capable of accommodating the rapidly evolving needs of even the most demanding radio system customers and users.

The driving force behind many software-based features however, is configuration. Whether it is the redefinition of talkgroups to accommodate an increased subscriber population, or the adjustment of talkgroup coverages to more intelligently govern access to scarce radio channels – software-based features often rely heavily upon dynamic configuration. Undeniably, software-based features are finding their way into more and more facets of PLMR systems – and in so doing – are dramatically increasing the importance and criticality of their associated dynamic configurations.

The focus of this paper then, is the robust design of these dynamic configurations. By robust, we mean that configuration mechanisms should be designed: (1) to be as forgiving as possible of disconnected and/or failed system elements; (2) to minimize the
penalties of errors and missing/incomplete data that cannot be ignored completely, and (3) to provide straightforward compensatory mechanisms for resetting, correcting, and counteracting unexplainable misbehaviours and long-lived error conditions not correctable in the short term. To illustrate the worthiness of robustness as a goal, imagine a PLMR system used in a public safety application. The ability of configuration to gracefully deal with a wide range of exceptions can be invaluable during times of emergency – a natural disaster, or a terrorist attack perhaps – when system infrastructure may only be partly survived and PLMR services are needed the most.

The outline of this paper is as follows. In Section 2, we briefly survey the challenges faced when designing configuration for PLMR systems. We argue that the large scope of PLMR systems introduces unique difficulties in the field as well as organizationally. Section 2 also acknowledges the fact that proper, user-centered design processes are often lacking in many PLMR system projects, and that robust configuration designs can help make such projects “more livable” in spite of poor user interfaces. Section 3 proposes and describes a series of guidelines to facilitate robust configuration designs, while Section 4 chronicles anecdotal cases in which those guidelines have been applied. Finally, Section 5 provides a summary and makes recommendations for future work.
Towards Robust Configuration of Advanced PLMR Systems

2 Challenges

Within the domain of land mobile radio, configuration design challenges stem from having wide-ranging coverage requirements, multifarious development organizations, and non-trivial system procurement processes that obscure the needs of users. Robust configurations can help to address such challenges and ultimately make the PLMR system more successful.

2.1 Remote Geographical Access

PLMR systems are often used by large organizations with extensive coverage requirements. This can mean equipment deployments in difficult-to-access areas such as deep wilderness regions or far-off mountaintops. Apart from the obvious barriers imposed by Mother Nature, organizational barriers – in the form of stringent security procedures – can also exist if equipment happens to be deployed in military or police controlled settings.

And so, whether it is several hours worth of mountain-pass driving, a day-long trek in a 4x4 jeep, or a week’s worth of security clearance paperwork – equipment in PLMR systems can be decidedly difficult (and slow) to gain access to. Add to the fact that large PLMR systems can have remote equipment sites easily outnumbering technician teams – and one can appreciate the fact that careful planning is needed in even the simplest deployment, upgrade, and/or repair scenarios.

Robust configuration designs can help by being generally tolerant of missing, out-of-date, or malfunctioning remote site equipment until technicians have the opportunity to address them. A robust, well-designed PLMR system should be capable of providing services in the interim, irrespective of the difficulties encountered by having a large geographical “footprint” of equipment. In addition, because geographically isolated sites are usually
exposed to the elements – they can be unpredictably affected by floods, hailstorms, lightning, power surges and power outages, etc. Hence, configuration mechanisms should also be designed to routinely endure periods of intermittent network connectivity.

2.2 Makeup of the Development Organization

To cope with size and complexity, PLMR system architectures are usually broken down into smaller subsystems to be developed separately and mostly in parallel. Hence, to develop a PLMR system necessarily involves the co-operative efforts of many software and hardware development teams working in conjunction. Precisely how individuals and responsibilities are assigned amongst these teams – i.e. the makeup of the development organization – plays no small part in the ultimate success of a PLMR system.

While hardware\(^1\) development teams are usually formed in line with the actual “physical” hardware pieces in the system, the forming of software development teams is often much less straightforward. By their very nature, software subsystems can be somewhat ambiguous, spanning across multiple hardware pieces and/or platforms. Normally, software development expertise is also harder to assess (and hence, to leverage) because it tends to be more technology-focused\(^2\) than product-focused. Consequently, when subsystems are mapped onto teams, software developers are more apt to feel estranged than their hardware counterparts do, working in unfamiliar technologies on unfamiliar subsystems. As pointed out in [2], often the “technical structure of the architecture shapes the social structure of the team and vice-versa” – which can lead to conflict as the “collective hopes, experiences, dreams, fears, and preferences” of disparate software development teams clash.

\(^1\) Hardware development in this case is assumed to include low-level DSP firmware development. Generally, any kind of development that is easily and unambiguously associated with hardware.

\(^2\) Hardware development team expertise is usually easily transferable from product to product. Software products on the other hand, can change considerably from generation to generation as operating systems continue to develop, and software development technologies continue to evolve. For example, when it comes time to assign the next generation of a software product to developers – competition can result between development teams that have historically worked on previous generations of the product and development teams that are more familiar with the newer operating systems and technologies (but may have correspondingly little knowledge of previous generations of the product).
Towards Robust Configuration of Advanced PLMR Systems

That being said, configuration mechanisms are essentially software subsystems – the design and implementation of which are very much influenced by the makeup of a development organization. Configuration subsystems have a high-level of interconnectedness – spanning multiple hardware pieces and platforms, and requiring the collective efforts of many development teams to implement correctly. Yet, configuration subsystems in particular tend to receive disproportionately little attention for something that can so drastically affect the final PLMR system. System-level issues (spanning across several configuration subsystems) are often overlooked as a result. For example, the disparate and competing priorities of individual development teams can lead to “unbalanced” configurations and strange configuration idiosyncrasies. All of these factors can contribute to a PLMR system that is much more difficult to configure than the “sum of its parts” might initially suggest.

By actively promoting robust configuration design guidelines within the development organization, there can be refocused awareness of system-level issues. Unbalanced interfaces become more “balanced” irrespective of the different paces exhibited by development teams, and the effects of configuration idiosyncrasies can be reduced as mechanisms become more tolerant of exceptions. Essentially, robust configuration design guidelines can help to provide a foundation for successful designs in the same manner as Postel’s Prescription [3] provided a foundation for the internet [4]; and how unique design philosophies provided a foundation for the creation and durability of the UNIX operating system [5].

---

3 We coin the term “unbalanced” to represent those configuration interfaces that are very accommodating for some development teams, while being bulky, unwieldy, and challenging for other teams. Although there are often a variety of other organizational factors, unbalanced interfaces often come about naturally because slower-to-develop subsystem teams simply do not/cannot afford the time and resources to fully comprehend the ramifications of intricate configuration mechanisms until it is too late. Ultimately, the robustness of the PLMR system suffers, as do delivery schedules.
2.3 System Procurement, Human Computer Interaction, and Safety Issues

A lack of user involvement – and consequently, incomplete user requirements – are often ranked as the top two reasons for unsuccessful software projects [6]. PLMR systems are typically procured using a formalized process centered around the Request For Proposal (RFP) – a business term referring to a request for bids, through a tender process, on a specific product or service (also known as an RFQ, request for quotation). By their very nature, RFP’s must purposely focus upon abstract system-level requirements, and are often uniformly weak in the area of understanding user requirements [7], though almost all PLMR system RFP’s contain loose statements about “user friendly interfaces”.

The gap between procurers and end-users is made worse by the fact that RFP’s are often issued by government agencies, which can be considerably far removed from actual radio system users. Even after a winning bid/proposal is selected, negotiations between government agency procurers and the PLMR system vendor rarely reach a level of detail where end-user needs are sufficiently examined [7]. Furthermore, end-users such as system configuration maintenance staff may not even be hired – much less formally trained upon – the PLMR system until long after it has been deployed.

The lack of user involvement described thus far often leads to user interfaces that are poorly matched to end-users – of which, the user interfaces responsible for maintaining the dynamic configurations in advanced PLMR systems are no exception. Dynamic configuration interfaces borne out of a lack of user involvement are commonly perceived as difficult to use because they pre-suppose an operator with the same expertise and in-depth understanding of design models as developers have [8]. Interfaces can even be construed as “dangerous” or “unsafe” if users can easily cause deleterious effects in the PLMR system during routine usage and exploration of the user interface. Ultimately, users can lose confidence and withdraw, claiming adamantly that their PLMR system is too complex or “fragile” to configure on their own. In extreme cases, formal acceptance of the entire PLMR system can be delayed almost indefinitely. Because such HCI and such safety issues arise so very late in the process – concessions can be very difficult to
negotiate. Not only is the PLMR system usually deployed by this point, but the
deficiencies found at such a stage are often deeply rooted – perhaps even going back to
how subsystems were originally mapped to development teams in the first place (as
discussed in section 2.2 Makeup of the Development Organization).

Ideally, user involvement in the RFP process would increase, and user interfaces would
be designed iteratively to better match the characteristics of those who must maintain the
dynamic configurations of advanced PLMR systems. However, the likelihood of either
occurring is quite slim – the RFP is a well-established process that is unlikely to reform
itself in the near future, while political and organizational factors often defeat attempts at
iterative user interface design. The recourse suggested in this paper then, is to instead
concentrate upon the robust construction of underlying dynamic configuration
mechanisms – to at least lessen the degree to which usability and safety issues can impact
final PLMR system operation.

By being generally forgiving of user mistakes, robust configuration designs can relieve
user anxiety levels enough to support the formation of better mental models through
exploration, even of poorly-matched user interfaces. More generally, robust
configuration mechanisms can be supportive of users as they: perform compensating
transactions; reassert the same actions during times of stress and emergency; and make
small mistakes without fear of causing big accidents; – all despite less-than-ideal user
interfaces. All of these are desirable behaviours suggested by HCI research into safety-
critical systems [7]-[10]. Robust configuration designs are in effect – attempting to
design in some of the “defensiveness” that [7] advocates for safety-critical user interfaces
– but within the configuration mechanisms themselves rather than at the user interface.
3 Robust Configuration Design Guidelines

In hopes of providing an adequate framework for the ensuing dialogue in section 4 Applying the Design Guidelines: Anecdotes and Examples, we choose to enumerate and present the design guidelines here first.

3.1 About Database Management Systems

Before going further, we should comment on the many relational DataBase Management Systems (DBMS’s) that come pre-packaged with built-in transaction facilities such as multi-phase commit, logging, automatic rollback, and data replication – all of which can be leveraged to build robust configuration mechanisms. Indeed, in the context of multi-tier, enterprise-class business networks – where DBMS’s are used throughout the network to automatically take care of data concurrency and synchronization issues – we accept that there may be a lesser need for explicit robust configuration design guidelines.

In the context of a PLMR system however, not all elements necessarily have the luxury\(^4\) – much less the computing resources – to run a full DBMS to store and manage their configurations, nor are DBMS packages necessarily available for those PLMR system elements running low-level, Real-Time Operating Systems (RTOS). Additionally, a wide variety of device-specific configuration formats tend to exist in a PLMR network (from proprietary, third-party vendor-specific formats to cryptic binary formats such as bit-masks and hex codes, etc). Adapting and unifying such formats simply so we can take advantage of DBMS facilities may be significantly prohibitive.

In fact, we should acknowledge and recognize that the database usage pattern within PLMR systems is often distinctly different from that of most enterprise-class business

\(^4\) Many network elements in a PLMR system have the majority of their computing resources (i.e. processor, memory, etc.) devoted expressly to DSP and trunking control algorithms.
Towards Robust Configuration of Advanced PLMR Systems

networks. [11]-[13] identifies and contrasts two typical database usage patterns, ApplicationDatabase, and IntegrationDatabase. When a database is predominantly controlled and accessed by a single application, it follows an ApplicationDatabase pattern. In contrast, when a database is controlled and accessed by multiple applications (thus integrating data across the applications), it is said to follow an IntegrationDatabase pattern. Because of application-specific requirements, and the need to provide basic functionality even when disconnected from the network, PLMR system elements typically follow the ApplicationDatabase pattern, while nodes within an enterprise-class business network typically follow the IntegrationDatabase pattern — sharing a large global database.

Our goal then, is to formulate robust configuration guidelines based upon concepts that can be both easily understood and practically applied across PLMR system elements — irrespective of whether DBMS's are used or not, and in keeping with the fact that PLMR system elements tend to follow the ApplicationDatabase pattern. Moreover, many developers may be in the midst of transitioning from hardware to software development methods and not necessarily fluent in computer science theory and/or database and DBMS concepts.

3.2 Some Brief Terminology

The following terminology is introduced to provide a framework for design guideline formulation and discussion. Again, keep in mind that our discussion takes place within the context of the ApplicationDatabase pattern [12], where each PLMR system element is assumed to control and access its own configuration database.
3.3 Guideline G1_TransactionStamp

**Description:** Consider two network elements \(E_1\) and \(E_2\) within the PLMR network with \(E_1\) providing configuration information towards \(E_2\). A mechanism should exist for \(E_1\) to become notified whenever \(E_2\) becomes “out-of-date” – requiring further configuration information from \(E_1\) to bring it up-to-date. Guideline G1_TransactionStamp advocates a simple mechanism, whereby \(E_2\) and \(E_1\) regularly exchange a shared scalar value transaction stamp \(V_{\text{trans}}\) to facilitate synchronization and resynchronization of their configurations. The guideline is outlined in detail following a pictorial representation in Figure 1.

![Diagram of G1_TransactionStamp](image.png)

**Figure 1.** Diagram of G1_TransactionStamp
Towards Robust Configuration of Advanced PLMR Systems

- For each transaction $T_{ij}(x)$ applied to $C_1$ that is propagated towards $E_2$, $E_1$ should formulate and associate a $V_{trans}^+$ with the transaction, and provide this to $E_2$.

- Upon being triggered by $E_1$ to bring its configuration up-to-date, $E_2$ should only persist the newly provided $V_{trans}^+$ if $C_2$ is successfully updated by the corresponding transaction $T_{2k}(y)$.

- The transaction $T_{ij}(x)$ performed on $C_1$, and the transaction $T_{2k}(y)$ performed on $C_2$ are related, but not necessarily the exact same transaction since $C_1$ and $C_2$ can each represent entirely different configuration schemas.

- We have kept $T_{ij}(x)$ and $T_{2k}(y)$ as single transactions to keep the guideline description simple. It is not hard to imagine however, cases in which $T_{ij}(x)$ and/or $T_{2k}(y)$ may instead represent multiple transactions.

- In addition, $V_{trans}$ should be implemented:
  - Such that it can be easily ordered using elementary “less than” ($<$), “greater than” ($>$) and equality ($=$) operations. Larger values of $V_{trans}$ should be associated with more up-to-date transactions, while smaller values of $V_{trans}$ should be associated with less up-to-date transactions.
  - Such that it possesses a large enough range to minimize wrap-around/overflow occurrences and to accommodate “stopgaps”. A stopgap being a large jump in the value of $V_{trans}$, intentionally introduced when $E_1$ is upgraded or repaired to ensure $E_1$ will be initiated with a value of $V_{trans}$ that is larger and more “up-to-date” than any existing $V_{trans}$ values held by $E_2$.
  - The exchange of $V_{trans}$ is not directionally-specific, and can either be based on a uni-directional mechanism whereby $V_{trans}$ is provided towards $E_1$ by $E_2$, $V_{trans}$ is provided towards $E_2$ by $E_1$, or a bi-directional mechanism that does both simultaneously.
- *G1_TransactionStamp* is essentially based upon a simplified version of timestamp-based concurrency control [14], with transaction stamps used to track when configuration items are changed, not when they are read\(^5\).

**Usage Notes:**
- *G1_TransactionStamp* should be applied whenever there are cooperating network elements that must synchronize their configuration information. Usually, a parent/child or master/slave relationship exists between such network elements (i.e. \(E_1\) provides configuration information towards \(E_2\)). In addition, a network element may be a child/slave to one element, but be a master/parent towards another element (i.e. though \(E_2\) receives configuration information from \(E_1\), \(E_2\) may in turn provide configuration information towards other elements \(E_3, E_4, \) etc).

- Many elements in a PLMR network may already exchange "heartbeat" messages with one another to stay current on connectivity status. Depending upon the situation and the size of \(V_{\text{trans}}\) used, \(V_{\text{trans}}\) may be a worthwhile payload to include in these heartbeat messages.

- By defining a clear association between \(V_{\text{trans}}\) values and transactions, not only can \(V_{\text{trans}}\) be used to detect non-synchronized configurations, it can also be used to help decide upon a subsequent configuration correction strategy. For example, if each increment of \(V_{\text{trans}}\) is deemed to represent a single transaction, then the calculated value, \(V_{\text{trans}, E_2} - V_{\text{trans}, E_1}\) can be taken to represent the number of transactions that the two network elements \(E_1\) and \(E_2\) differ by. A threshold applied to this calculated value can thus be used to decide upon the best resynchronization strategy to bring configuration on \(E_2\) up-to-date (i.e. a small number of additive transactions if the difference is small, and a relatively more involved resynchronization mechanism if the difference is large).

\(^5\) The correctness of read operations within PLMR systems often do not matter as much as they do in traditional database theory. This is because at any given time, portions of the radio system can be unreachable, in maintenance or repair mode, out of coverage, etc. In PLMR systems, it is far more important to guarantee that up-to-date operations are eventually read – rather than unnecessarily aborting transactions due to inappropriate reads – or worse, causing unnecessary resynchronizations. In the same way that voice-decoding errors are largely temporal, the effects of reading aged data in PLMR systems are similarly temporal in nature.
- *G1_TransactionStamp* simply states that some sort of transaction stamp should be used to detect non-synchronized configurations, and purposely does not dictate a resynchronization policy. Resynchronization policies may be immediate, delayed, or otherwise, depending on the application and the situation. In practical, day-to-day PLMR system operations, there may be times when resynchronization may be delayed on purpose (i.e. large difference in $V_{trans}$ during peak system operation hours), and other times when resynchronization can be done immediately because it is mostly harmless (i.e. small difference in $V_{trans}$ during off-peak hours).

### 3.4 Guideline G2_ConfigPropagation

**Description:** Consider a PLMR network element $E_o$ that is simultaneously a child to element $E_p$ and a parent to element $E_c$. Configuration changes on $C_p$ trigger configuration changes on $C_o$, which in turn trigger configuration changes on $C_c$. For configuration changes to propagate in a robust manner, any trio of elements within the PLMR network that can be modeled as $\{E_p, E_o, E_c\}$ should follow a consistent policy to disseminate configuration changes from $E_p$ through $E_o$ to $E_c$. Guideline G2_ConfigPropagation basically advocates a single-hop-at-time persist and propagate strategy where $E_p$ propagates configuration to $E_o$ irrespective of $E_c$, and $E_o$ in turn propagates configuration to $E_c$ irrespective of $E_p$. The guideline is outlined in detail following a pictorial representation in Figure 2.
For every set of nodes that can be modeled as \( \{E_p, E_o, E_c\} \) ...

... the propagation of configuration from \( E_p \rightarrow E_c \) should persist configuration at \( E_o \) first, before attempting to persist configuration at \( E_c \).

![Diagram of G2_ConfigPropagation](image)

**Figure 2. Diagram of G2_ConfigPropagation**

- For each transaction \( T_{p}(x) \) applied at \( C_p \) that must be propagated towards \( E_o, E_p \) should apply and persist associated \( T_{o}(y) \) upon the local \( C_o \) first, before attempting to further propagate configuration changes towards \( E_c \).

- Once the change is persisted in \( C_o' \), \( E_p \)'s role in the configuration propagation process is finished. \( E_o \) (not \( E_p \)) takes care of further propagating towards \( E_c \) to trigger an associated \( T_{c}(z) \) affecting \( C_c \).

- Missing, disconnected, or malfunctioning \( E_c \) should not hinder the ability of \( E_p \) to propagate changes towards \( E_o \). In other words, \( E_o \) essentially "insulates" \( E_p \) from the nuances associated with configuring \( E_c \).

- As before, the transactions performed on each of \( C_p, C_o, \) and \( C_c \) are related, but not necessarily the exact same transaction since \( C_p, C_o, \) and \( C_c \) can each represent entirely different configuration table schemas.

**Usage Notes:**
- \( G2\_ConfigPropagation \) should be applied whenever there are cooperating network elements that propagate information to one another in a multi-level
parent/child arrangement as depicted in Figure 2. More specifically, any set of elements in the network that can be modeled as \( \{E_p, E_o, E_c\} \) should propagate their configuration in the manner described by the guideline.

- Though \( G2\_Config\_Propagation \) seems trivial, the combined benefit (i.e. if all possible sets of elements modeled as \( \{E_p, E_o, E_c\} \) behave as per the guideline) is that configuration changes initiated from higher-level parent elements will naturally tend to reach the furthest available/functional child elements in the PLMR network. In addition, missing/malfunctioning elements in the PLMR network will automatically be updated when they come back online. This works because \( G2\_Config\_Propagation \) requires the immediate parent network element (i.e. \( E_o \)) to always persist and propagate configuration changes to \( E_c \), regardless of when it received the configuration from \( E_p \) originally.

### 3.5 Guideline G3_BatchAndDelta

**Description:** Consider a set of configuration tables \( C_i = \{C_{i1}, C_{i2}, ..., C_{iN}\} \) on element \( E_i \) that altogether require updating with one or more transactions to bring the entire set’s status to \( C_{i updated} \). Essentially, two different methods can be used to update \( E_i \)’s configuration, a “batch” method (which receives and buffers \( C_{i updated} \) in its entirety before updating \( C_i \)), and a “delta” method (which updates \( C_i \) to \( C_{i updated} \) via a series of incremental transactions). \( G3\_BatchAndDelta \) advocates that all PLMR network elements should strive to support both distinct methods of configuration processing. As before, the guideline is outlined in detail following a pictorial representation in Figure 3.
Towards Robust Configuration of Advanced PLMR Systems

"Batch" Configuration Processing

$E_i$

- In “batch” configuration processing, $E_i$ must first receive and buffer an entire set of changed configuration tables $C_i^{\text{updated}} = \{ C_{i1}^{\text{updated}}, C_{i2}^{\text{updated}}, \ldots, C_{iN}^{\text{updated}} \}$. Once $C_i^{\text{updated}}$ has been received in its entirety, $E_i$ completely supplants $C_i$ with $C_{i1}^{\text{updated}}$, $C_{i2}^{\text{updated}}$, $\ldots$, $C_{iN}^{\text{updated}}$. Whether $E_i$ chooses to overwrite the original $C_i$ with $C_i^{\text{updated}}$, or if $E_i$ instead maintains two copies of configuration and simply switches between the two is largely implementation-specific.

- “Delta” is considered the opposite of batch configuration processing, and instead changes $C_i$ to $C_i^{\text{updated}}$ through a series of incremental transactions $T_{ij}(x)$.

- Without a proper delta configuration processing method, batch configuration processing is “overkill” for smaller configuration changes, wasting network bandwidth, and perhaps unnecessarily disrupting PLMR system stability. On the other hand, batch methods are much more efficient when larger configuration changes and error-recovering re-initializations must be carried out. Hence, to appropriately support different PLMR configuration activities with minimum

---

6 The term “batch” originated in the days when users entered programs on punch cards [15].
disruption to PLMR system services, \textit{G3\_BatchAndDelta} advocates that elements should strive to provide both batch and delta configuration processing methods.

\textbf{Usage Notes:}

- To be effective, batch implementations should ensure that bulk configuration information can be received in a "buffered" manner, without affecting current configuration and operation of the network element. Applied across the entire PLMR system, it can help to ensure uninterrupted availability of PLMR system services. Buffering also facilitates integrity checking of the entire configuration once received. We should be wary of batch implementations without buffering capabilities, as they tend to be nothing more than long-running delta-configuration processing methods in disguise.

- The decision to perform a batch or delta configuration update can easily be based upon a thresholded \( V_{\text{trans}} \) difference (as discussed previously in \textit{G1\_TransactionStamp}).

\subsection*{3.6 Guideline G4\_InformationHierarchy}

\textit{Description:} The administration of thousands of talkgroups and subscriber devices are often altogether handled as part of advanced PLMR system configuration. To make such large configurations easier to manage, tables are often normalized\(^7\) to reduce storage requirements and to minimize redundancy and repetition of data. Normalization usually results in hierarchical tables linked together by foreign keys.

A foreign key is a column in one table that refers to (or targets) a specific column, usually the primary key, in another table. A primary key can be targeted by multiple foreign keys from other tables, but does not necessarily have to be the target of any foreign keys. A

\footnote{\textsuperscript{7} "Delta" is contrasted against batch in the same spirit that "online or interactive" processing was contrasted against batch processing when users moved away from punch card systems, towards time-sharing systems \cite{16}.}

\footnote{\textsuperscript{8} There are many database normalization forms that are defined and often discussed (see \cite{17}-\cite{20} for example), but we are referring here to one of the most commonly applied "defacto standard" normalization forms – the Third Normal Form (3NF) – which basically ensures that each table contains unique data.}
foreign key relationship implicitly creates a hierarchy between two tables – the table containing the foreign key column is referred to as the child (or dependent), and the table containing the primary key from which the foreign key values are obtained is the parent. Figure 4 illustrates the foreign key concept.

Table B

<table>
<thead>
<tr>
<th>primary key</th>
<th>data columns</th>
<th>foreign keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>&lt;x's data&gt;</td>
<td>z</td>
</tr>
</tbody>
</table>

Table A

<table>
<thead>
<tr>
<th>primary key</th>
<th>data columns</th>
<th>foreign keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>w</td>
</tr>
<tr>
<td>y</td>
<td>&lt;y's data&gt;</td>
<td>z</td>
</tr>
</tbody>
</table>

Table C

<table>
<thead>
<tr>
<th>primary key</th>
<th>data columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>&lt;y's data&gt;</td>
</tr>
</tbody>
</table>

Table D

<table>
<thead>
<tr>
<th>primary key</th>
<th>data columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>z</td>
<td>&lt;z's data&gt;</td>
</tr>
</tbody>
</table>

Figure 4. General Illustration of Foreign Keys

Database relationship diagrams such as the one in Figure 5 are another common way of representing foreign keys:
Now, with any use of foreign keys, one must give due consideration to the topic of referential integrity. A database (or in our case, a set of configuration tables) is said to possess referential integrity if, for every foreign key value, there is a matching primary key value in an associated table row. Continuing on this train of thought, we coin the term “transaction cascade” to represent a sequence of transactions that are constrained to follow a certain order for the express purposes of preserving referential integrity. For example, in the context of Figure 4, the possible transaction cascades associated with the deletion of item z in table D are:

1) \{Delete A[w], Delete B[x], Delete D[z]\}
2) \{Change A[w], Change B[x], Delete D[z]\}
3) \{Change A[w], Delete B[x], Delete D[z]\}
4) \{Delete A[w], Change B[x], Delete D[z]\}

where:
- \(<table name>[primary key]\) refers to the row residing in table name at primary key.
- the Change operation is assumed to alter the foreign key reference to something other than item z (in table D).
Having established the concept of foreign keys, referential integrity, and transaction cascades, we can now go on to define guideline \textit{G4\_InformationHierarchy} as the following:

- \textbf{G4\_InformationHierarchy.1} Carefully weigh the trade-off between: 1) the need to enforce referential integrity with 2) the subsequent increase in interface complexity that results from having to communicate using transaction cascades. Keep in mind that in the context of a PLMR system, each element typically hosts its own configuration database, so the complexity of transaction cascades can quickly add-up as elements propagate configuration between one another.

- \textbf{G4\_InformationHierarchy.2} If referential integrity must be enforced, consider enforcing it at higher-level PLMR system elements that are responsible for the configuration of many child elements, rather than enforcing it at each lower-level child element. Such a strategy can help keep complexity to a minimum, and better support future growth and flexibility in the referential integrity constraints.

- \textbf{G4\_InformationHierarchy.3} If referential integrity must be enforced, ensure that the complete set of elementary add, change, and delete operations are available for all data items. This minimizes the information cascading that is required if the data item becomes used as a foreign key.

- \textbf{G4\_InformationHierarchy.4} If referential integrity must be enforced, ensure that the removal or changing of foreign key references is a simple orthogonal operation (i.e. does not affect other rows in the table, and does not affect other columns in the row). Previously in Figure 4, the foreign key values $y$ and $z$ could both be removed orthogonally from Table A and Table B. Comparatively, Figure 6 below illustrates non-orthogonal uses of foreign key value $z1$:
Towards Robust Configuration of Advanced PLMR Systems

Usage Notes:

- Experiences with existing PLMR system projects have shown that structuring information hierarchically can and does help maintenance staff to plan, maintain, and organize their configurations. For example, creating re-usable radio personalities that can be "shared" across an entire population of subscribers is usually well received by users. A hierarchical representation of radios and personalities saves users from having to define and maintain a separate personality for each and every radio if they in fact share the same personality parameters. To re-iterate, G4 _InformationHierarchy_ does not discourage structuring information hierarchically – it in fact advocates it – but with the qualification that not all information hierarchies require strict referential integrity constraints.
Consider the consequences that referential integrity checking will have not only at the system level, but for users as well. For example, consider again the case of reusable radio personalities. One user may wish to create subscriber units first, assigning empty, but named personalities in the process, and then filling in the personalities later. Instead, another user may wish to fully define all personalities first, and subscriber units later, assigning them the personalities he/she created previously. Yet a third type of user (probably the most realistic of the three) may casually alternate his/her workflow between both working styles, depending on his/her mood, and the information provided to him/her from others. In this case, applying extensive referential integrity constraints on radios and radio personalities would tend to limit users to the second working style only.

As mentioned before, although DBMS's provide built-in referential integrity checking, not all elements within a PLMR system necessarily have the resources available to run a DBMS. Moreover, even those PLMR system elements that can afford to run a DBMS often find that built-in DBMS features do not fit well with application-specific requirements. The reality then, is that most, if not all of the referential integrity checking in PLMR system elements ends up being performed by application code, and is hence more prone to software defects. Referential integrity checking also tends to occur naturally between PLMR system elements, and therefore susceptible to defects caused by misunderstandings and general lack of coordination between development teams. Both of these vulnerabilities should be carefully taken into consideration when considering whether to enforce referential integrity or not.

The enforcement of referential integrity is a frequently discussed topic in software engineering. Recent debates argue that referential integrity is better enforced on a case-by-case basis, rather than strictly enforced all the time by DBMS packages (and the like) [21]. Moreover, editorials such as [22] remark that the ability to handle "dangling links" (i.e. broken foreign keys) naturally leads to more robust systems that function even when partially survived and are evolvable over time as requirements change.
4 Applying the Design Guidelines: Anecdotes and Examples

4.1 Replacing Connectivity-based Synchronization

On one PLMR system project, two critical infrastructure elements were using a simple connectivity timer to maintain synchronization. Element A functioned in a parent/master role, providing configuration towards element B, which functioned in a child/slave role. Essentially, as long as elements A and B continued to receive periodic heartbeats from one another within $T_{disc}$ seconds, each element assumed that it was sufficiently synchronized with the other. The connectivity-based synchronization mechanism is shown in Figure 7 (compare this to the earlier $G1_{TransactionStamp}$ diagram).

![Figure 7. Connectivity-based Synchronization](image)

Major problems were experienced with the connectivity-based synchronization mechanism:

- Whenever element A or element B ran for greater than $T_{disc}$ without the reception of heartbeats, the two elements would need to re-synchronize with one another, even if no configuration items had changed between the two.
Towards Robust Configuration of Advanced PLMR Systems

- A variety of factors could trigger a cessation of heartbeats between elements A and B. Sporadic network congestion, lightning storm induced power outages, software upgrades, element A being too busy to heartbeat towards element B (or vice versa), and even routine rearrangement of equipment racks holding elements A and B. As discussed previously in section 2.3 System Procurement, Human Computer Interaction, and Safety Issues, we wish to avoid system behaviours that give users the impression that the system is "fragile" with "small changes leading to big accidents" [7].

- Upon reconnection, PLMR system services would be crippled while parent element A resynchronized child element B. Depending on the size of the configuration, the resynchronization could take several hours.

- When examined individually, element A and element B were each considered to be behaving correctly – each performing as best it could given the fleeting frame of reference that connectivity provided. Parent element A would essentially lose faith in the configuration held by child element B when heartbeats from B ceased for more than $T_{disc}$, henceforth directing element B to resynchronize upon re-establishment of communications. Likewise, child element B followed a similar policy of caution – seeking resynchronization with parent element A whenever heartbeats ceased for more than $T_{disc}$.

The problem here was that the connectivity mechanism simply did not provide a robust frame of reference, and was highly sensitive to even the most minute network disruptions – precisely what is addressed by GI_TransactionStamp. There was more to the problem than just network connectivity however.

As mentioned before, the makeup of the development organization can pose a significant challenge to developing PLMR configuration systems. In this particular case, the fact that parent element A was developed in a completely different office than element B indeed perpetuated the problem. That the two development teams were separated by
combination of geographical and organizational barriers made communication difficult and hindered a mutual understanding of the issue. The lack of understanding led to workarounds such as extending the $T_{\text{disc}}$ interval to very large values, and elaborate planning to perform upgrades in less than $T_{\text{disc}}$ seconds. Ultimately, the root of the problem— that "connectivity" was a poor frame of reference to use in the first place— eluded both development teams for much longer than necessary.

Applying $G1\_\text{TransactionStamp}$ eventually resolved the issue because it addressed both technical problems as well as the difficulties associated with the makeup of the development organization. By focusing upon a common understanding of the guideline, the two development teams were able to bridge relationships, and make significant inroads across geographical and organizational barriers. As a result, software changes were successfully enacted on both element A and element B to support the proper implementation of $G1\_\text{TransactionStamp}$. Moreover, not only did $V_{\text{trans}}$ provide a much better frame of reference for resynchronization, it could also be shared, understood, and discussed easily across the development teams when debugging.

### 4.2 Gracefully Applying Large Scale Resynchronizations

In the next case, a failure to differentiate between batch and delta configuration processing made large-scale resynchronizations very inefficient and slow, to the point of being disruptive. The result was again PLMR service outages lasting hours at a time depending upon the size of the configuration. Any time extended service outages occur, a customer's confidence in their PLMR system can diminish significantly. Now, unlike the previous case described in section 4.1 Replacing Connectivity-based Synchronization the resynchronizations in this case were not unexpected, and in fact, were providing a valuable global system reset facility. Instead, the problem in this case was rooted in the inefficiency of the configuration processing mechanisms associated with the resynchronization.
Central to this case was a voice switch infrastructure element that plays a critical role in providing PLMR voice services. The voice switch element implemented two methods of receiving configuration from its parent element, referred to as "entire reload" and "delta reload". Though the "delta reload" method closely followed $G3_{BatchAndDelta}$'s recommended delta configuration processing mechanism, the "entire reload" method was far from a true batch configuration processing mechanism. Upon closer examination, the "entire reload" was nothing more than a global "clear-all-configuration" operation followed by an overly-extended "delta reload" as depicted in Figure 8a. The larger the configuration, the more the "entire-reload" would inefficiently over-extend the "delta-reload", resulting in the long service outages that we opened this discussion with. Figure 8 compares and illustrates service outages by showing the subscriber population-per-time profiles of (a.) an over-extended "delta reload"; (b.) a hybrid method that at least performs buffering of the over-extended "delta reload"; and (c.) proper batch configuration processing as recommended by $G3_{BatchAndDelta}$. 
Towards Robust Configuration of Advanced PLMR Systems

**a. Over-extended “Delta Reload”**

- Over-extended "delta reload" grows linearly with size of configuration
- Subscribers must re-register with the system
- Over-extended “delta reload” configuration

**b. Over-extended – but buffered “Delta Reload”**

- Over-extended – but buffered “delta reload” configuration
- Subscribers must still re-register with the system

**c. Proper Batch Configuration Processing**

- Subscribers do not need to re-register with system
- Buffered receive
- Configuration switch-over

---

Figure 8. Over-Extended “Delta Reload” vs. Hybrid Method vs. Proper Batch Configuration Processing
Towards Robust Configuration of Advanced PLMR Systems

To achieve the profile shown in Figure 8c required software changes on both the voice switch element as well as the element sending the configuration. Due to time and organizational constraints, changing the sender's delivery mechanism was not viable.

However, as a proof-of-concept, and to facilitate migration towards a complete implementation of batch processing, the behaviour of the voice switch element was modified such that it at least "buffered" the contents of over extended "delta reloads". This resulted in a subscriber population-per-time profile depicted in Figure 8b. In this hybrid method, even if the contents were not being delivered in a bulk fashion by the sender, a much improved situation was nonetheless realized simply because voice services continued to be available while buffering the "entire reload". Also notice that the service outage time is no longer linearly dependent on the size of the configuration being resynchronized. So, even with just a hybrid method, resynchronization operations had become much more graceful and much less disruptive. Currently, the proof-of-concept hybrid implementation is still being evaluated. If accepted, there will be additional motivation to modify the sender to deliver its configuration more efficiently, and for the voice switch element to retain subscriber registrations across reloads.

4.3 Isolating Configuration Propagation Effects

As discussed in section 2.1 Remote Geographical Access, the likelihood of missing, disconnected, or malfunctioning elements is quite high in PLMR systems. Recall that G2_ConfigPropagatiorz advocates that configuration changes should propagate in a step-wise manner towards the furthest reachable, operational elements. Now, consider the parent/child configuration relationships of two groups of PLMR system elements (again based upon system elements from an actual PLMR project) depicted in Figure 9.
In the case represented on the left of Figure 9, missing or malfunctioning PLMR site elements tended to delay responses all the way back at the front-end configuration GUI. This was problematic because users at the configuration GUI only possessed a very high-level overview of the entire PLMR system that was devoid of specific PLMR system elements. In other words, there were very little (if any) explicit references to the voice switch element at the configuration GUI – much less any references to the PLMR site elements for that matter. Users of the GUI would quickly lose confidence in their actions because the delays would not only interrupt their workflow activities, but the delays would occur unexpectedly, without warning, and without meaningful error and/or warning messages.

In the case represented on the right of Figure 9, a different problem presented itself – users were completely prohibited from making any changes to configuration if even one of the voice switch elements was missing and/or unreachable. Such behaviour was very disconcerting to the customer because it meant that if a disaster wiped out one of the voice switch elements, they would have to take steps to explicitly “decommission” the destroyed voice switch element – just to restore the ability to configure the remaining voice switch elements.
In each case, first reactions involved attempts to better train and educate the users in the nuances and behaviours of the PLMR system. Such attempts were largely unsuccessful because in the first case, the random nature of errors would still interrupt workflow regardless – while in the second case, users tended to reject the notion of having to decommission the voice switch element as a workaround (the voice switch is a critical element responsible for providing PLMR voice services and non-trivial to remove from the configuration GUI). Furthermore, even if users were trained in the nuances of the system, the relatively infrequent nature of configuration activities, coupled with the random nature of the errors, would tend to work against any real knowledge retention.

Both cases can be recognized as a violation of $G2_{\text{ConfigPropagation}}$. In each case, users are being unnecessarily exposed to system error conditions that are hierarchically inappropriate. According to $G2_{\text{ConfigPropagation}}$, users should instead be allowed to delegate authority to the local GUI application, and not worry about the error conditions and nuances experienced by system elements at subsequently lower levels.

Using this technique, the first case has since been resolved by redesigning the GUI to store user changes locally. Then, on behalf of the user, the GUI application is responsible for automatically configuring and reconfiguring the voice switches beneath it, and then each voice switch is in turn responsible for automatically configuring and reconfiguring the site elements beneath it. Delays and error conditions at each successively lower level are effectively insulated from the higher levels above. Hence, site elements in general are now handled silently and automatically by the voice switch rather than causing random delays and interruptions to user activities at the GUI level like before.

The second case has yet to be fixed on a PLMR system project, but can be solved in a similar manner. The GUI should be redesigned to store user changes locally. Then, on behalf of the user, the GUI application should be made responsible for automatically configuring and reconfiguring the voice switches beneath it as they come and go. Implemented in such a manner, users at the GUI would no longer be prevented from making configuration changes simply because one or more voice switches were non-operational or unreachable.
4.4 A High-Availability System

Many times, the guidelines are applied in conjunction with one another, rather than in isolation. Such was the case when the configuration of a high-availability “pair” of PLMR infrastructure elements was redesigned. Before the re-design, all configuration changes from an external configuration GUI were initially passed towards the secondary peer – based on the premise that the secondary peer could serve as a “test candidate” for all configuration changes before being applied to the primary peer. The idea was novel, but was problematic in practice. The rather unusual propagation mechanism led to strange and difficult-to-trace error conditions where changes would succeed on the secondary, but fail on the primary, warranting the need to implement elaborate undo mechanisms between the primary and the secondary. Even worse, if a configuration change went through on the secondary, but caused the primary to shut down, the remaining secondary peer could potentially remain forever out-of-sync (by one transaction) with the configuration GUI – with no apparent detection mechanism to catch the idiosyncrasy.

The existing design also employed a connectivity-based synchronization mechanism similar to that described in section 4.1 Replacing Connectivity-based Synchronization – along with all its associated problems. Just like our previous discussion, the high-availability pair was overly prone to resynchronizations from occasional heartbeat message losses.

The new design was made simpler and more robust by first following both \textit{G1\textunderscore TransactionStamp} and \textit{G2\textunderscore ConfigPropagation}. Replacement of the connectivity-based resynchronization with \textit{G1\textunderscore TransactionStamp} also provided a basis for peer-to-peer batch and delta resynchronization mechanisms to be implemented according to \textit{G3\textunderscore BatchAndDelta}. After the redesign, not only was the high-availability pair less prone to resynchronizations, each peer could now decide intelligently – when to perform smaller delta resynchronizations and when to perform larger batch resynchronizations – simply by comparing the difference in $V_{ran}$, between themselves. A summary representation of the design before and after is given in Figure 10.
After the redesign, developers benefited from having a more consistent framework to write code within, which resulted in an ~14% reduction in Lines of Code (LOC), even including other features added during development. In addition, the step-wise configuration propagation mechanism also turned out to be more scalable because it could now accommodate multiple standby peers opening up future possibilities for cluster-type high-availability designs to replace the current pair-based design.
4.5 Add/Delete as a Shortcut for Change

A "shortcut" sometimes taken by developers is to provide only add and delete operations on configuration items, and to forego providing an explicit change operation. The rationale being, that the effects of a change operation can easily be "mimicked" by performing a delete operation followed by an add operation. This sounds simple enough, but can lead to quite awkward usability issues if the configuration item is used as a foreign key (i.e. referred to by other configuration items). To illustrate, consider once again Figure 4 from the G4_InformationHierarchy discussion, and imagine the "transaction cascade" necessary to change a single field in $D[z]$ if referential integrity is enforced and table D only allows Add/Delete (i.e. no Change is allowed):


where:
- \(<table name>[primary key]\) refers to the row residing in table name at primary key.
- the Change operation is assumed to alter the foreign key reference to something other than item z (in table D).

In fact, the tricky "transaction cascade" scenario just described occurred on a PLMR system project and sets the stage for an anecdotal discussion on applying G4_InformationHierarchy.

In this case, the problem presented itself after the PLMR system was already deployed, and just as system maintenance staff began to train on the system (recall the discussion in section 2.3 System Procurement, Human Computer Interaction, and Safety Issues). Spurred by a user request to change the name of a talkgroup within the system, a tangled, transaction cascade on a much larger scale than the one just presented was uncovered.
On this PLMR project, the lack of a “change-talkgroup” operation meant that simply changing the name of a talkgroup required that all configuration items that referred to the talkgroup be temporarily removed and then re-entered into the system. Users even regressed to primitive means such as pencil and paper to jot down information that needed to be re-entered. The workaround wreaked havoc on user’s mental models ([8]) of the system, as no user could understand why something as simple as a talkgroup change could be so involved. Users began to wonder whether all changes were so involved. Again, the incident likely diminished the confidence of the system maintenance staff, and deterred their willingness to explore ([7]-[10]).

So ironically, what began as a well-intentioned shortcut by developers turned into a rather disastrous “long-cut” – manifesting itself as a system level usability issue late in the PLMR project. Applying \textit{G4-InformationHierarchy} could have avoided the problem since it recommends that all three of the elementary add, delete, and change operations be implemented whenever there is the possibility of a configuration item being used as a foreign key.

Though the PLMR system in this scenario has since solved the problem by adding the missing change operation, there still remain many configuration items that use foreign keys in a non-orthogonal manner (another part of \textit{G4-InformationHierarchy}). These non-orthogonal uses of foreign keys can be considered just as bad as the missing change operation because they too can lead to complex transaction cascades and ultimately usability problems.

\subsection*{4.6 Over-Applying Referential Integrity}

\textit{G4-InformationHierarchy} recommends that referential integrity checks should be applied sparingly, and need not be applied to all configuration databases incorporating the use of foreign keys. Consider two cases taken from PLMR system projects, illustrated in Figure 11. In the case on the left, Configuration GUI A is responsible for propagating configuration towards multiple voice switches, while in the case on the right,
Configuration GUI B is responsible for propagating configuration towards a single voice switch.

In each case, the voice switch element’s configuration consists of hierarchically related tables, with heavy use of foreign keys to minimize data duplication. The voice switch enforces the fact that foreign key references must remain valid at all times, so that configuration GUIs must send configuration data in a referentially consistent manner, and extensive “transaction cascading” (as discussed in G4 InformationHierarchy) becomes necessary. Consequently, a considerable amount of software complexity in the form of “referential integrity logic” becomes necessary – at the voice switch (which checks all incoming configuration for referential integrity violations) and in each of the configuration GUIs (each of which must re-shuffle and re-cascade outgoing configuration transactions to satisfy referential integrity constraints).

As one can see, development of referential integrity logic must be carefully coordinated between the voice switch and each of the configuration GUIs – especially if each is the
responsibility of a different development team. Moreover, maintaining the coordination between so many development teams as the voice switch configuration schema evolves over time can prove equally – if not more challenging.

In this particular case, configuration GUI A, configuration GUI B, and the voice switch were in fact, each developed by a different development team. The development teams for the voice switch and configuration GUI B were lucky enough to be co-located within the same office, while the development team for configuration GUI A was geographically and organizationally separated from the other two development teams.

Coordination across teams proved quite difficult. For instance, consider that only configuration GUI B could initially configure the voice switch element while configuration GUI A could not. Of course, this scenario put pressure upon the development team responsible for configuration GUI A, even though it was not really a fair comparison. For one, configuration GUI A was designed to work at a PLMR system level, dealing with the configuration of multiple voice switch elements, while configuration GUI B was expressly tailored for the configuration of a single voice switch element (and in a much better position to make design-simplifying assumptions). The team responsible for configuration GUI A was also working at a significant disadvantage because they were geographically and organizationally isolated from the developers who worked on the voice switch element. Again, one can see the impact that the makeup of a development organization can have.

As time went on, frequent software defects resulted from the voice switch’s strict enforcement of referential integrity coupled with the lack of coordination amongst the three development teams. These defects were a serious matter for the customer because the voice switch is deemed as one of the most critical elements within a PLMR system. To address the problem, G4_InformationHierarchy was applied and the voice switch’s need for referential integrity was closely re-evaluated. Detailed examination revealed that the voice switch was essentially putting up a “referential integrity façade” and that most of its internal mechanisms were quite pessimistic, making no assumptions of referential integrity anyways. By removing the “referential integrity façade”, and making
the voice switch software universally tolerant of broken foreign key references, coordination between the voice switch and the two configuration GUIs would no longer need to be so tight. Slight mis-coordinations between the three development teams still led to software defects, but such defects did not “cripple the PLMR system outright” as they would have before referential integrity constraints on the voice switch were relaxed.

Additional benefits were also realized. Developers of the voice switch enjoyed a reduced mental load, because they no longer had to maintain the referential integrity façade, struggling with inconsistent code that enforced referential integrity in some areas while not enforcing it in others. Software complexity in the voice switch was also significantly reduced — a LOC savings of ~13% was realized (a remarkably similar savings to that described in 4.4 A High-Availability System). Presently, configuration GUIs continue to check for referential integrity at the user interface, but such checking is free to evolve and change, no longer constrained by a tight coupling to the voice-switch’s referential integrity scheme.

In the context of [5], this scenario can be viewed as a case where referential integrity constraints were overly applied (premature optimization), resulting in GUI tools that were too closely bound with the voice-switch’s referential integrity checks (a failure to separate policy from mechanism).
5 Conclusion

Dynamic configurations are providing the foundation for software-based features in revolutionarily flexible, advanced PLMR systems. Such configurations should be carefully designed to be robust – i.e. tolerant and forgiving of errors, but easily compensated and adjusted when errors do occur. Robustness can help to address the various challenges faced by PLMR configuration design: difficult-to-access field equipment; complex development organizations; and lack of user involvement during system specification and deployment.

A simple way of achieving robust PLMR configuration designs is to actively define, promote, and follow, simple design guidelines across a PLMR development organization. In this paper, we described a series of four such guidelines, and then illustrated their application in a series of anecdotal examples derived and abstracted from real-world PLMR system project experiences. The anecdotal examples revealed that guidelines could be applied together, alone, or even partially to evaluate and solve many technical and organizational issues related to dynamic configurations in advanced PLMR systems. Moreover, the examples also brought to light the fact that many PLMR system usability issues have dimensions and aspects that lie beyond just the user interface – which all too often serves as a convenient scapegoat.

That the application of such a small number of robust configuration design guidelines were so successful is encouraging, and suggests that further work in this area is worth pursuing. For example, a comparison study of the guidelines described in this paper to commonly used robust design patterns in network design, object-oriented design, etc. may yield important insights into why the guidelines work, as well as suggest candidates for other potential guidelines.
The work in this paper also suggests that user-centered design processes indeed look beyond the user interface itself. It may be worthwhile for example, to survey actual PLMR system users in an attempt to characterize where the majority of usability problems in PLMR systems lie. Subsequent post-analysis of the survey data could then aim to reveal whether typical PLMR system usability issues are as deeply-rooted as the ones described in this paper, or whether most usability issues are instead localized just to user-interfaces.
References


