Tolerating Interaction Faults Originated From External Systems

Bogdan T. NASSU† and Takashi NANYA†

† Research Center for Advanced Science and Technology (RCAST), The University of Tokyo
4-6-1 Komaba, Meguro-ku, Tokyo 153-8904, Japan
E-mail: \{bogdan,nanya\}@hal.rcast.u-tokyo.ac.jp

Abstract This work introduces a new scenario and a fault model for tolerating interaction faults. This issue becomes more critical as monolithic systems give place to systems composed by other systems, designed by independent parties, and which employ communication standards to interact. If one of the systems implements the standard in an incomplete or incorrect manner, interaction faults may occur. The problem of designing a system able to tolerate faults using only local knowledge about the standard is addressed. We present a general architecture for error detection and correction, based on traditional techniques and the concept of implicit redundancies.

Key words Interaction Faults, Communication Protocols, Fault Models, Implicit Redundancies

1. Introduction

More and more, systems built from interconnected systems become present in people’s everyday lives. The Internet is already spread worldwide, and ubiquitous computing[1], the integration of computers with the environment, is quickly becoming a reality. These systems are not monolithic blocks: they may be designed by different parties, at different moments, in different conditions, and following different practices. Interaction faults are a major issue in this setting.

Independent systems employ communication standards to interact. The standard can be, for example, a file format which both systems understand, or a protocol such as TCP/IP[2]. However, sometimes one of the systems does not implement the standard correctly, or implements it only partially. This may occur, for example, if the specification of the standard was misinterpreted, or if there was insufficient debugging or testing. In those situations, interaction faults may occur when the systems try to communicate.

The problem addressed in this paper is: how can a system tolerate interaction faults originated from an external system, without changing the communication standard or the faulty system itself? A detailed description of the problem, including a fault model, is presented. We also discuss the general mechanisms that can be used to detect and correct errors, including a concept called implicit redundancies.

The remainder of this paper is organized as follows. In section 2., the system and fault models are defined. In section 3., some considerations about other scenarios and related work are made. In section 4., the general mechanisms for error detection and correction are presented. Finally, section 5. concludes the paper and points to future work.

2. System and Fault Models

This section presents a model for the proposed scenario, as well as a fault model. Three important concepts, tasks, communication patterns and context, are also defined. We begin by describing a situation that exemplifies the scenario, considering the Universal Plug-and-Play[3] architecture.

2.1 A Scenario Example

Universal Plug-and-Play (UPnP)[3] is an architecture used for communication between different types of electronic devices, including computers and home appliances. UPnP allows a device to announce and offer services to a control point, and is based on several well-known protocols, such as IP, HTTP and XML; as well as some specialized protocols.

Suppose a consumer owns a DVD player that uses UPnP to interact with the TV set. For example, the TV can adjust its own configurations based on the disc inserted in the DVD player. If the UPnP implementation in the DVD player is faulty, interaction faults may occur. The consumer does not know the source of the problem — all he sees is that the complete system does not work. He could even blame the manufacturer of the TV for the problem. If the TV set is designed to tolerate faults originated from other devices, this situation could be avoided.

2.2 System Model

The system is defined as a pair of interacting sub-systems. Interaction occurs through a communication standard, with the exchange of a sequence of messages, each containing a set of data fields. The system that can exhibit faulty behavior is called the external unit, or E. The other system, which is designed to tolerate faults, is the adaptable unit, or A.

To isolate the problem, it is assumed that, apart from the
implementation of the standard, \( E \) is fault-free. That includes lower level protocols needed by the considered standard, which are abstracted as a fault-free “communication interface”. Thus, \( E \) can perform without error functions that are not related with the communication standard. We also assume that all faults originate from \( E \) — designing an unit capable of tolerating its own faults is a different problem.

Figure 1 shows a system instance, considering the example presented in section 2.1. The adaptable unit is the TV set, which is a UPnP control point that uses the services offered by the DVD player — the external unit. As the considered standard is UPnP, the communication interface is not only the physical link between the units (e.g. wireless communication), but the entire TCP/IP stack below UPnP.

![System example considering UPnP.](image)

2.3 Scenario Peculiarities

This model has some important peculiarities, which will limit the approaches that can be used for fault tolerance.

- **The presence of an external unit.** As the external unit is not part of the unit being designed, the designer of the adaptable unit has no control over the faulty implementation of the standard. Moreover, the top-level system is only instantiated when the units communicate — i.e. the specific external unit is not known while the adaptable unit is being designed. Thus, error detection and correction must be done in runtime, while the units interact.

- **The communication standard has been already specified.** That means no changes can be made in the standard, as the external unit cannot be required to implement them.

- **There is only a pair of units.** As the model assumes an interaction between two units, the adaptable unit cannot rely on other units besides itself to detect and correct errors.

2.4 Tasks, Communication Patterns and Context

To manage the complexity of the standard, it must be divided smaller parts. Though this is usually done by the means of state machines, in this work we use a broader concept: tasks. A task is a collection of states working towards a single purpose in the standard. Each task is defined by its semantics (its meaning inside the standard), the type of information it deals with, and a communication pattern. The communication pattern describes the sequence of messages; and may be represented by regular expressions [4], with the symbol \( a \) denoting a message sent by the adaptable unit and \( e \) a message sent by the external unit. Each task occurs inside a given context. The context is defined by the system instance (i.e. the external unit to which the adaptable unit is connected to) and the instant the task occurs.

The definition of the boundaries for a task is subjective and standard-dependent, but most specifications already describe the standard in small units, such as “end connection” or “request file”. The division in a few complex tasks; or in a large number of small tasks with complex relationships; should be avoided. A good measure of the complexity is the communication pattern — it should not involve a great number of conditional branches, hard to express as regular expressions. Common communication patterns include \( ae \) (a request/response cycle), \( (ae)+ \) (a series of request/response cycles) and \( e+ \) (several incoming messages with no reply).

2.5 Fault Model

As said in section 2.2, all faults originate from \( E \), and only when the units interact. A fault occurs when \( E \) exhibits a behavior that deviates from the specification of the standard (the specification is assumed to be correct); or when the data inside a message is incorrect. The faults can be analyzed according to the classes presented in [5]. As \( A \) is the unit being designed, the classification is done assuming its viewpoint:

- **Major Class: Interaction Faults.** As the faults are caused by a flawed implementation of the standard, they could be mistakenly seen as design faults. Indeed, for the external unit, they are design failures (the manifestation of design faults), but for \( A \) they are interaction faults.

- **Phase of Creation or Occurrence: Operational Faults.** The faults will manifest for the adaptable unit only when the system is operating and the units try to communicate.

- **System Boundaries: External Faults.** The faults are all generated by a system external to the adaptable unit.

- **Dimension: Not Applicable.** The adaptable unit makes no distinction between faults caused by buggy software and faults caused by bad hardware design choices.

- **Phenomenological Cause: Human-Made Faults.** The faults are the result of human errors.

- **Objective: Non-Malicious Faults.** It is assumed that the external unit means no harm to the adaptable unit.

- **Intent: Non-Deliberate Faults.** It is assumed that the external unit does not exhibit a faulty behavior intentionally.
• **Capability:** Not Applicable. For the adaptable unit, it does not matter if the faults are accidental or the product of a designer/programmer’s incompetence.

• **Persistence:** Permanent. The flawed implementation of the standard cannot be expected to be corrected in the short term. A repaired $E$ should be seen as a new unit.

When an interaction fault occurs, it may result in one or more errors. As $A$ does not have direct access to the flawed implementation of the standard, the errors are described in terms of what can be observed outside $E$. Each error can be seen as falling into one of several categories, as shown below:

- **Incorrect Action:** $E$ performs an unexpected action.
- **Incorrect Data:** is divided in three sub-categories:
  - **Missing Data:** a field inside a message is missing.
  - **Obviously Incorrect Data:** a field has an unacceptable value for its data type.
  - **Not Obviously Incorrect Data:** the value for a field is acceptable for its type, but is incorrect.
- **Formatting:** the external unit sends an ill-formatted message. Note that this type of error refers to the format of the message, not of data inside data fields.
- **Sequence:** the external unit sends messages or performs actions out of the expected sequence.
- **Timing:** the external unit performs the expected actions and sends eventual messages, but too early or too late.
- **Omission:** the external unit does not send a message it was expected to send, or sends it with a very long delay.

Considering a single error, the error categories overlap and relate to each other as shown in figure 2. The following relationships can be highlighted:

- All errors can be seen as particular cases of incorrect actions. “Pure” incorrect actions are the ones which cannot be observed as other types of errors.
- Formatting errors are a particular case of incorrect data, where data is present but is not properly formatted.
- Omission errors are a particular case of missing data, incorrect sequence, and incorrect timing: all data is missing, and the message is received after infinite time, breaking the expected sequence.

### 3. Related Work

The problem of tolerating faults originated outside the boundaries of a system is not new. A great number of protocols incorporate robustness schemes which allow communication to be successful even in the presence of faults. Some of these solutions depend on cooperation from both units — so they may not work if the implementation is faulty in the external unit. There are also strategies based only knowledge local to one of the units. For example, TCP [2] uses timeouts and sliding windows to deal with omission and sequence errors. Another example: in real-time transmission of voice over the Internet [6], an unit can use buffers to deal with variable delays and generate substitutions for lost packets.

However, the problem presented in this paper was previously addressed aiming at specific applications and at the design of protocols. Our scenario provides a more general model, which enables new schemes to be added to the adaptable unit, besides the ones defined by the protocol. In fact, changing the protocol is usually the best way of dealing with the problem, in a technical sense. However, updating the standard in existing systems is often not feasible, for economic and practical reasons. Furthermore, our model works under the assumption that the protocol implementation may be faulty in one of the units — a real issue which is not taken into account by most protocol designers.

Other related work include conformance and interoperability testing [7]. These are techniques used to check if an implementation conforms to the protocol specification, and if a given unit works correctly when interacting with another unit. They are used during the design stage, to detect and correct bugs. However, in our scenario, the aim is not removing design faults from an unit, but to design an unit able to tolerate interaction faults originated from an external unit.

Differences can also be observed between this work and techniques for the design of dependable user interfaces [8, 9]. Both scenarios deal with faults originated from an external entity, but dependable user interfaces aim at avoiding faults. This is achieved by changing the way information is presented to the user, i.e. the protocol. In our scenario, the communication standard cannot be changed; and the faults cannot be avoided — they have to be tolerated.

### 4. Tolerating Faults

To effectively tolerate $E$’s faults, the adaptable must employ a series of error detection and correction schemes. Even
if these schemes are not able to detect or correct every single occurring error, they are a valuable tool if they can provide a reasonable coverage. Before the schemes are defined, it is important to recall the restrictions imposed by the system model’s characteristics, as said in section 2.3:

• Error detection must be done online.
• The communication standard cannot be changed.
• The adaptable unit cannot rely in other units.

These restrictions limit the choice of mechanisms for error detection and correction. For example, it is not possible to use some traditional techniques [10], such as unit duplication, temporal redundancy (retries), or offline test phases, unless the standard already provides some kind of “test mode”.

In this section, we show how an adaptable unit can tolerate faults originated from an external unit. First, the concept of implicit redundancies is explained. Then, general mechanisms for error detection and correction are described. An architecture for an adaptable unit is also presented. The section also addresses the limits of the proposed approach.

### 4.1 Implicit Redundancies

Though “traditional” techniques can be used to address several types of errors, in some cases errors can only be detected or corrected with the “smart” employment of information available in the specification of the communication standard. This information may refer to the problem domain or the structure of the protocol itself. For example, applications working with real time transmission of voice over the Internet [6] employ receiver-based error correction schemes based on the understanding of how people interpret sound.

This type of receiver-based scheme was previously employed only for specific applications, or was embedded in the specification of the standard. In this work, we generalize the concept to the notion of **implicit redundancies**. An implicit redundancy is defined as any information which is not originally meant to be used to support fault tolerance, but which can be employed in that sense. This notion appears from the observation that several high-level protocols include some “entropic” information, which gives a “hint” about what can be expected in the future, or even is duplicated in different stages of the communication.

### 4.2 Detecting Errors

To tolerate faults, the adaptable unit must be first able to detect the errors which result from them. The different types of errors presented in section 2.5 require different error detection schemes, as shown below:

• **Timing and Omission**: these types of error refer to the receiving of a message inside some time constraints. The difference between timing and omission errors is that timing errors usually refer to a shorter amount of time. In fact, it is known that a late message can be indistinguishable from a lost message, and several communication standards consider them to be the same. Timing and omission errors are traditionally detected by timers and timeouts, with the specific values being defined by the task and context.
  • **Sequence**: each task has its own communication pattern, which represents the expected sequence of messages. Sequence errors can be detected by simple tests, comparing the order of the messages with the communication pattern, and by checking fields such as counters or sequence numbers.
  • **Formatting**: formatting errors are errors in the format of a message, not the data within it. Message formats in general can be described by context-free grammars [4], or even simpler rules. Thus, this type of error can be detected by format parsers, which can be automatically generated from the specific format employed by a task.
  • **Missing Data**: missing data errors can be detected by simple tests, which check if the required data fields are present inside a message. The required fields are defined by the task and the context (e.g. information received in previous messages, or other fields inside the same message).
  • **Incorrect Data**: the general strategy for detecting both obviously incorrect and not obviously incorrect data is comparing the received data with “references”. However, it is not possible to describe these references in detail, because they will be different for each type of data: they can be a set of if/then statements, a simple range check, a mathematical function, or even a value to be directly compared to the received data. To detect obviously incorrect data, references are usually created based on static information, available from the specification of the standard and data type. Not obviously incorrect data errors can be harder, or even impossible to detect, as they refer to data which is normally accepted by the specification of the standard and data type. To detect not obviously incorrect data, references depend on the availability of implicit redundancies, specially values received in previous tasks and messages.
  • **Incorrect Action**: “pure” incorrect action errors have no visible effect in the messages received by the adaptable unit. For that reason, they can be very hard, or impossible to detect. Some incorrect action errors result in specific patterns of behavior, such as cycles or repetitions. These patterns can be detected by probabilistic algorithms based on implicit redundancies.

### 4.3 Correcting Errors

After an error is detected, the adaptable unit may try to correct it. The correction scheme may be different for each type of error, as shown below:

• **Timing**: if there are rigid real-time requirements — i.e. if there is no possible tolerance for early or late messages — timing errors can be considered as omission errors.
Otherwise, the adaptable unit may extend the timeout to a larger value, up to a given limit (chosen according to the task). If this tolerance is not enough, the message should be considered lost. Even if this happens, messages which are received late should be analyzed, and any useful information (i.e., implicit redundancies) should be saved for later use.

- **Sequence**: sequence errors are traditionally addressed with techniques which hold the out-of-order messages until the other expected messages are received and the sequence is re-established. This can be done, for example, with the use of sliding windows [2].

- **Formatting**: correction schemes for formatting faults can vary significantly from format to format, and should be decided in a case-by-case manner.

- **Omission and Incorrect Data**: When a message or field is missing or incorrect, the adaptable unit must choose a course of action. There are three possible strategies:
  - The adaptable unit may mask the faulty data, reconstructing the message or replacing fields. To this purpose, it may use constant (default, or otherwise “harmless”) values and implicit redundancies, learned from previous tasks or messages.
  - If the faulty message is a response to a request, the adaptable unit may change parameters to make it more general, accepting more responses (including the expected one).
  - The adaptable unit can trigger other tasks, which will perform a role complementary or similar to the current task.

Determining the replacement values to be used, the parameters to be changed or the tasks to be triggered will depend on the semantics of the task and the context. The same can be said about choosing the best course of action.

- **Incorrect Action**: though the same strategies used for omission and incorrect data may apply, correction schemes for this type of error may differ significantly depending on the error and the context of its occurrence. We define no general means of correcting this type of error.

### 4.4 General Architecture

The proposed schemes for error detection and correction must be included in the adaptable unit, as an extension to any robustness mechanisms defined by the standard. Figure 3 shows the general architecture for a fault-tolerance mechanism which can implement the proposed schemes. The following components are present in the architecture:

- **Function**: this block includes the usual implementation of the communication standard, as well as other functions, unrelated to communication. The Function block can be seen as A without the fault tolerance mechanism.

- **Task Knowledge**: there are several blocks of this type, each holding the specific knowledge necessary to detect and correct errors in a single task, including static information and implicit redundancies. The Task Knowledge block is implemented by data objects and program code, and is further detailed ahead.

- **Knowledge Base**: this block can be seen as a library containing the set of all Task Knowledge blocks.

- **Error Detector (General)**: the Error Detector (General) is responsible for the error detection schemes which are the same for all the tasks. This include timers for timing and omission errors and format parsers for formatting errors.

- **Error Corrector (General)**: the Error Corrector (General) is responsible for the error correction schemes which are the same for all the tasks. It also provides an interface between the specific error correction schemes in the Task Knowledge block and other parts of the system.

- **Request Generator**: this component is used whenever the adaptable unit sends a message to the external unit. In normal operation, the message is simply forwarded, with some information being copied to the Task Knowledge block. In cases where error correction involves reissuing requests, the Request Generator is employed to create a message without affecting other functions in the system.

It can be seen that the Task Knowledge blocks centralize several essential functions in this architecture. They are the ones responsible for everything that differs from task to task. Figure 4 shows a more detailed view of a Task Knowledge block. The following sub-blocks are present:

---

![Figure 3](image1.png) **General architecture for an adaptable unit.**

![Figure 4](image2.png) **Internal view of a Task Knowledge block.**
• **Task Variables**: this block is responsible for keeping information about the context of the task. It holds the implicit redundancies learned during the task execution.

• **Task Constants**: this block holds static information about the task. These constants include, for example, lists of required fields, value ranges and default values.

• **Error Detector (Task)**: these are the task-specific schemes used to detect missing and incorrect data errors.

• **Communication Pattern Tracker**: this block keeps track of the received messages, and checks how they fit in the task’s communication pattern. It detects sequence errors and activates timers in the Error Detector (General) when messages are expected.

• **Error Corrector (Task)**: this block contains the task-specific schemes used for correcting omission and missing/incorrect data errors.

### 4.5 Discussion: Limits of this Approach

The proposed architecture and schemes for error correction and detection provide a general view of how an adaptable unit can be implemented. However, while some some of the schemes are traditional or very simple (e.g. detecting timing errors), others lack a detailed description. More specifically, this can be observed in the schemes for detecting incorrect data and action errors; and correcting missing/incorrect data, omission, formatting and action errors.

The fundamental obstacle for the creation of detailed and universal fault tolerance schemes is the adopted fault model. Though it is able to describe the errors that occur due to interaction faults, it is very abstract, and describe some types of errors only in general terms. That occurs because the semantics of each task, message and field are relevant for this problem. This is radially different, for example, from models which consider inverted bits [10]. In that case, the model does not depend on the semantics, and errors can be specified in detail for anything that can be represented as a stream of bits — no matter which type of data these bits represent.

Given these restrictions, it can be seen that the proposed architecture cannot be evaluated as it is. It is not possible to determine how well errors can be detected, which errors can be corrected, or which is the performance cost for \( A \). Therefore, the proposed models should be used as a starting point for the creation of more specific versions of the problem, considering individual communication standards. The fault tolerance architecture can be seen as a general framework, with the specific schemes being defined according to the protocol, its tasks, and the types of data involved.

### 5. Conclusions and Future Work

In this paper, a new scenario for addressing interaction faults was introduced. In this scenario, an adaptable unit is designed to tolerate interaction faults originated from an external unit. The faults occur because the external unit has a flawed implementation of the communication standard.

The main difference between this and existing work is that the problem was previously addressed aiming at specific applications and the design of protocols. Our scenario is more general, and enables new fault tolerance schemes to be added to \( A \), besides the ones defined by the protocol. Furthermore, our model works under the assumption that the protocol may be ill-implemented in one of the units — a real issue hardly taken into account by protocol designers.

General error detection and correction mechanisms were described. They are based on a combination of traditional techniques and the notion of implicit redundancies. These schemes depend on the semantics of each task and data type. Thus, this problem should be described following a general model, but must be addressed in a case-by-case manner.

The proposed models must be further explored, being employed in a specific case. This will enable us to create detailed error detection and correction mechanisms, and explore the concept of implicit redundancies. We are currently working on the specific case of the UPnP architecture. This includes the implementation of an Internet gateway device that can exhibit faulty behavior and a fault-tolerant control point.

**Acknowledgement**: This work was motivated in part through discussions in the collaboration with Matsushita Electric Industrial Co., Ltd.

### References