Simple Error-Correcting Communication Protocol for RCX

Pavel Petrovic, ppetrovic @t acm.org
Department of Computer and Information Science
Norwegian University of Science and Technology
Trondheim, Norway.

Technical report 3/2006.
ISSN: 1503-416X.

Abstract

We are currenly living the age of communications. Information revolution implies efficient and reliable means of communication. Most of the communication and information storage media are not error-free, while the requirements on the digital information often do not tolerate any errors in transmission or storage retrieval. To cope with this contrast, efficient error-correcting codes and error-detecting codes were developped in the second half of the last century. This paper summarizes our experience with implementing an error-correcting protocol for communiction of PC with LEGO RCX running Lejos (Java environment) over unreliable infrared link coupled with radio BlueTooth transmission.

Keywords: Error correcting codes, Erasure codes, LEGO RCX, Lejos, Communication

Introduction

This work was initiated through a student project focusing on safety and security analysis of rescue mobile robot operating in a building site. The robot prototype was built using LEGO construction set with programmable autonomous microcomputer RCX [2]. The robot navigates through the site. With the help of wireless network camera and image processing algorithm running on remote workstation it identifies victims within the site. The secondary objective is to attempt mapping of the area that is free for navigation. The robot is sending position information to the workstation and receiving optional control commands when it is remotely-cotrolled.

RCX normally communicates with the workstation using embedded infra-red (IR) serial port (running on 2400 bps, 8 bits, odd parity, and 1 stop bit). The range of the IR signal is limited and the communication requires direct or indirect (in the case of walls being reflective enough) visibility between the workstation and the robot. This is obviously not necessarily the case when exploring a building rescue site.

In order to extend the communication range and reliability, the robot was equipped with bi-directional IR to BlueTooth (BT) converter [1].

The requirement of the student project was to use the Java programming language and thus the only reasonable choice for programming the RCX was

the Lejos programming environment [9]. Unfortunately, in the version of Lejos that was available to us, the IR communication was not completely reliable, and some of the bits were flipped during the transmission. According to the ommunication settings, each byte was bundled with a parity bit, causing the incorrectly transmitted bytes to be completely discarded by the BlueTooth module in the cases when the parity did not hold. On the other hand, we observed that the transmission errors occured only occasionally, and did not tend to affect consecutive bytes, more specifically, in a series of four bytes, we could always find at most one of the bytes to be the subject of an error. Since the project requirement was to transmit the data correctly, we set on to design and implement an error-correcting protocol for our settings.

The following sections briefly introduce the field of Error Correcting Codes (ECC), and present and discuss our simple solution to the problem.

Short Introduction to Error Correcting Codes

Error correcting codes deal with transmitting digital information encoded in bit sequences. The goal is to encode the original message in such a way that even if some bits are transmitted with an error, the receiving side can recover the original message without having to request the message to be resent another time. The errors can appear individually, or in "bursts" of several consecutive bits - thus the algorithms either work bitwise or bit-tuplets-wise (for example bytes).

In the most simple solution, which takes the assumption of maximum one error occurence, each piece of information is sent three times, and the decoding side uses simply the majority vote to decide the original code. For instance, the byte 0x0d would be transmitted as 0x0d, 0x0d, 0x0d, and could arrive as 0x0d, 0x42, 0x0d. The receiving side will determine 0x0d being the orignal message. In this case, the encoding tolerates an error of up to 8 bits, however, they may not be spread around more than a single byte. If the position of the 8-bit error burst could be arbitrary, two of the bytes could be affected, and thus for the majority voting method, 5 bytes would have to be sent.

Fortunately, there are more space-efficient encodings for error correction supported by the theoretical properties of numbers and algebraic structures (finite fields and groups). The most famous one being used also for storing data in compact discs, hard-drives, and for communication with distant space missions, was discovered a couple of decades before the technology allowed its large-scale use: Solomon-Reed encoding [3]. The basic idea is that a message of certain length N is encoded into a message of slightly longer length m, where each element in the target sequence depends (polynomially) on all elements of the input sequence. Since it is possible to recover the coefficients of a polynom of degree N from N points, it is also possible to recover the original sequence from a subset of N correct elements of the target sequence. As we do not know which of them are correct, the majority vote applies, and thus the encoding can correct up to $(m - N) / 2$ errors. For instance, it is possible to encode the original sequence of 251 bytes as a sequence of 256 encoded bytes, where two of them might still be transmitted with error.

Other type/application of correcting codes deals with correction after removal - when some data at known locations are missing. This often occurs when a lower level of the communication protocol discovers an error in the transmission using a checksum or CRC, and throws that particular data packet away. The *erasure-correcting* codes are able to recover this missing information when the lower level indicates the location of the missing data. Clearly, the same idea as above can be used, and the redundancy can even be lowered, since no majority voting is needed as the location of the error is given. See [6] for an example and discussion of use of such forward-correcting codes for networking applications.

For more details about the various error-correcting codes, background theory and applications, we refer the reader to two excellent books [4,5].

The communication

The Lejos system is using the RCX's ROM routines for sending and receiving data through the serial infra-red port. The ROM routine sends each byte *B* of the IR packet as two bytes B, and ~B - this is to generate a stream of data that has 50% of bits equal to 1. ~B is the byte B with all its bits inverted. The ROM routines require that the incomming messages are compatible with the Mindstorms Robotics Invention System opcodes (instruction codes). Most instruction codes have some particular meaning - i.e. turn on motor, move value to a variable, etc., but a specialized instruction (opcode 0x45) allows sending a message containing data. These can be saved into a memory buffer provided by the user Lejos program running on the RCX. In result, in order to transmit a message containing only three bytes 0x01,0x02, and 0x03, the whole message packet of 25 bytes: "0x55, 0xFF, 0x00, 0x45, 0xBA, 0x00, 0xFF, 0x00, 0xFF, 0x04, 0xFB, 0x00, 0xFF, 0x03, 0xFC, 0x01, 0xFE, 0x02, 0xFD, 0x03, 0xFC, 0x06, 0xF9, 0x55, 0xAA" has to be transmitted. It consists of a message header, opcode, length, two checksums, and each byte is sent also as inverted. Still, we did not encounter any problems with reliability - except, perhaps, when a message that is longer than the size of the buffer provided by the Lejos user program gets accidentally received (for instance during the debugging of an application), this results in RCX lockout, and several-minute firmware download, because the remaining bytes simply overwrite Lejos data or program. The communication is not symmetric, and the other direction is more simple. No need for opcodes, thus a shorter messages can be transmitted. Unfortunatelly, the Lejos/ROM sending routines do not work perfectly, and sometimes emit erroneous IR signal. We believe that this is due to the fact that Lejos firmware does some interrupt-handling for multitasking, or another low-level system activity that interferes with the sending routines, but the outcome is that the messages are sometimes sent with bit errors. Since the IR signal is received with the IR to BT converter, the BT module simply discards those bytes where the parity bit check fails. As a consequence, the message received by the BT receiver on the side of the PC does contain occassional erasures at unknown locations as well as occassional errors - in the cases that the erroroneous byte still passed the parity-bit checking. Finally, in all our measurements, we noticed that the scrumbled or missing bytes occur seldom - never more often then every fourth byte. These were the observations that we took as the starting point for our solution.

Solution to the encoding problem using finite state automaton

Most of the error and erasures treatment in the literature deals either with error correction, or with erasures at known locations. Our aim was to design a simple encoding system that could handle erasures at unknown locations as well as errors. For the scale of the project was limited, we also had to find a solution that was easy and fast to implement and still worked reasonable enough.

Consider, for instance, the simple three-times-each encoding. If we send a sequence of bytes, there is a problem, because if one of the bytes is missing, even though the majority vote decides on the correct result for the first tripple - the next tripple is already shifted and thus a single error will lead to three bytes, each of them being different.

In addition, our possibilities are restricted by the fact that the ROM routines are sending each byte doubled (an original byte followed by an inverted one). On the other hand, we can benefit from this redundancy.

Each packet of data in our encoding is preceded by a header of three `0xFC` bytes (and their inverted versions). Our first attempt was to send each byte of the packet twice (i.e. 4 bytes `x,~x,x,~x`). However, two cases - when `x`, or respectively `~x` are lost in the transmission, are ambiguous. The two situations: `~x,x,~x` versus `x,~x,x` have the same pattern `a,~a,a`. Our second attempt for solution was to send the second byte incremented by one, resulting in the transmission, `x,~x,(x+1),~(x+1)`. There are, however, still several unplessant situations, in particular when in the original data `x` is followed by:

1. `~x`,
2. `(x+1)`,
3. `(~x + 1)`, or
4. `~(x+1)`

In such situations, ambiguities may arise when some byte is lost or modified (see the discussion). We therefore introduced a special escape character (`0xF3`), and encoded the above four situations as `x,ESC,0x71`; `x,ESC,0x81`; `x,ESC,0x91`; and `x,ESC,0xA1`, respectively. The ESC character is then coded as `ESC,0x41`, the start-header character as `ESC,0x31`, and their inversions as `ESC,0x61`; and `ESC,0x51`. Since everything is sent as `x,~x,(x+1),~(x+1)`, we need to protect also against the "`x+1`" versions, that is `0xF2` as `ESC,0x11`, and `0xF4` as `ESC,0x21`. Moreover, each packet is terminated by a sequence of several 0xFB bytes, which always reset the state machine to the starting state accepting the first byte of header, and thus also the byte `0xFB` is translated to `ESC,0x36`, and consequently its inversion `0x04` (as `ESC,0xC1`), its predecessor `0xFA` (as `ESC,0xD1`), and its inversion `0x05` (as `ESC,0xD1`). Finally, the code `0x7F` creates ambiguities, because `~0x7F = 0x7F + 1`.

Decoding the received message to obtain the original data is performed using an augmented finite state automaton, which is shown in the figure 1. The FSA can recover the data given at most one erasure or modification error occurs within each four bytes of the transmitted data. Transitions between states

correspond to bytes of the decoded input, whereas the output is constructed by the "`out var`" actions on the transitions. The FSA is also augmented with three variables `x, y, z`, which makes it contextual, and these variables can be used in additional predicates restricting the transitions, and in action assignments that are executed when the transitions are followed. An analogical FSA for decoding the packet header (which tolerates even some more errors in some cases) is shown in Figure 2.

The encoding is used only in the direction RCX -> PC (the other direction is reliable), and each packet received by the PC results in an acknowledge packet sent back to the RCX. The API on both sides provides the methods `send()`, and `receive()`. The `send()` method on the side of the RCX waits for the confirmation from the PC. If the confirmation does not arrive within a time limit, the method returns 0, and the application protocol can attempt to send the message again. Since the RCX ROM routines support short outgoing packets only, the packets that contain more bytes are automatically split into multiple packets, each of them being confirmed with an acknowledge message. The splitting happens without the user knowing about it - one `send()` method call can result in multiple packets being transmitted and acknowledged.

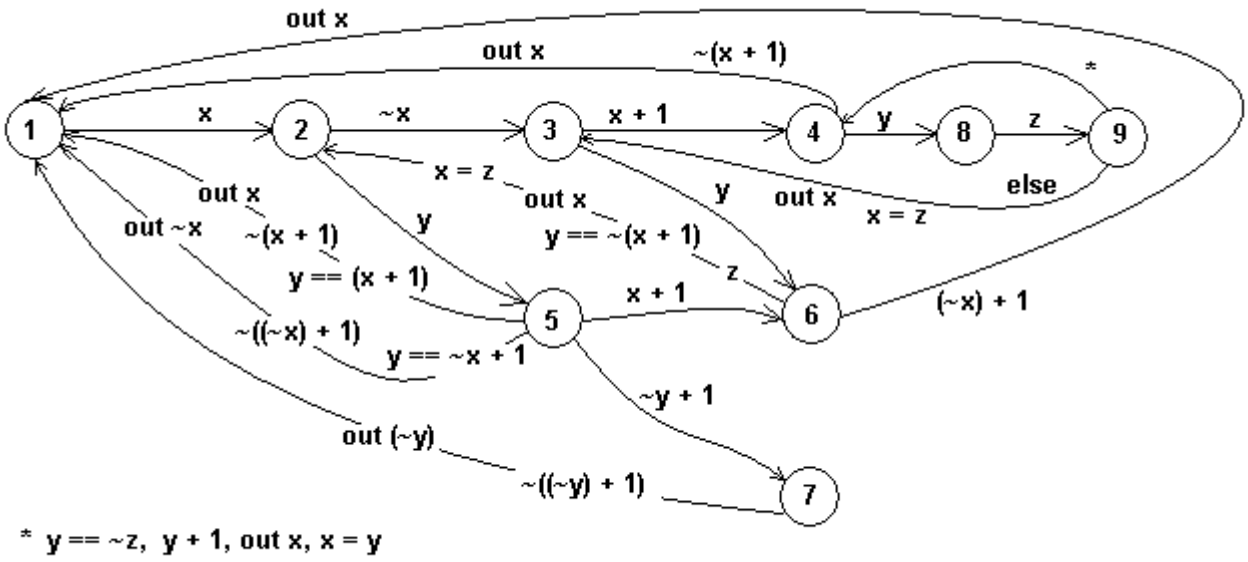

Figure 1. Decoding augmented FSA. `~x` corresponds to a byte with all bits of `x` inverted. The starting state is number 1. All states can be terminating states. The transitions in this compressed figure are marked by four types of labels: 1) *matching* - the transition is followed, if the next byte can be matched against the expression (such as `x`, `~(x+1)`), 2) *predicates*, (e.g. `y == ~(x + 1)`), which must hold for the transition to be followed, 3) *decoded bytes* (e.g. `out ~x`), and 4) *assignments* (e.g. `x = z`), which are necessary when the automaton loops back to a state that already assumes some variable matching.

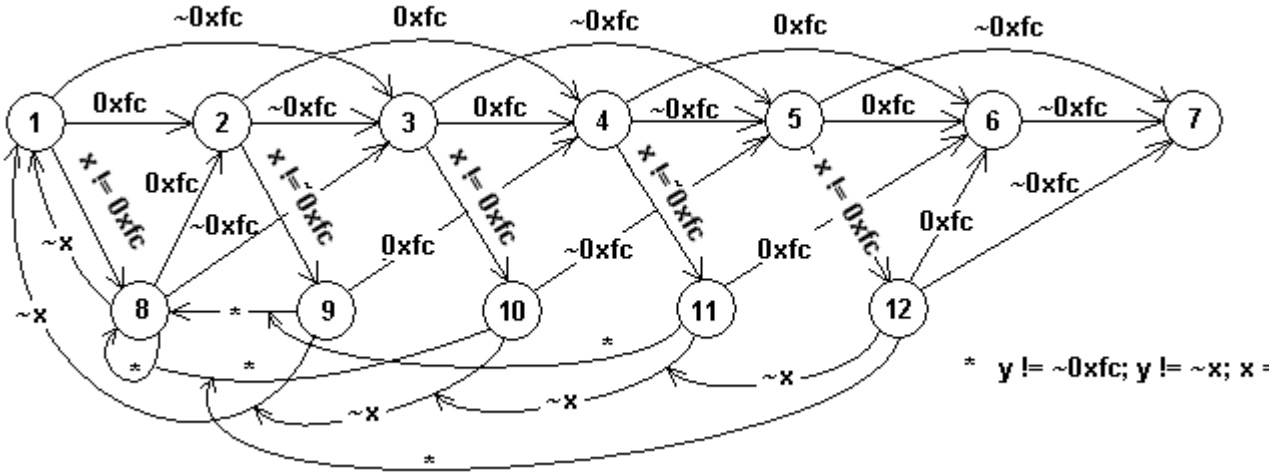

Figure 2. Decoding packet header. About labelling: For instance, transitions labelled `x != ~0xfc` are followed when any byte different than `~0xfc` arrives. This byte can in later transitions be referred to as `x`.

The early testing of the communication in the development lab showed a reliable transmission with virtually zero errors after the message has been decoded. However, as the battery level of the BT communication module decreases, the recognition capabilities of the time-inaccurate IR modulation produced by RCX with Lejos decreases, and the error frequency increases over the expected level, thus making some of the message packets to fail. This, however is still detected by the checksum errors, and when a message is not confirmed within a timeout, it is transmitted again, thus the receiving peer beeing able to recover from the transmission error. The sourcecode of the implementation is freely available at [7]. For the future projects, we recommend to use the C-based BrickOS system [8] instead of Lejos, as it seems that it communicates perfectly without any problems, and does not require any error correction protocol.

Discussion

We chose to implement a solution based on a hand-made finite-state automaton because the conditions did not match a standard easy error-correcting algorithm, and also for the reason of simplicity of implementation. What is missing is a proof that the encoding-decoding algorithm is complete, either using a formal proof or exhaustive testing. Since the longest path in the FSA has the length 4, which corresponds to the original sequence of length 1, it should be sufficient to try all 2^16 combinations of two consecutive bytes (transmitted sequences then will have the length of 8 bytes), and all the possible combinations of erasures and errors. We have tested exhaustingly all the possible combinations of two consecutive bytes against erasures at all 8 possible locations, and against modifications at all the possible locations, replacing by all the possible byte values, always getting the original message recovered. In addition, we provide the following discussion.

The automaton should correctly recognize the following 9 possible cases:

1. `x, ~x, x + 1, ~(x + 1)`
2. `y, ~x, x + 1, ~(x + 1)`
3. `x, y, x + 1, ~(x + 1)`
4. `x, ~x, y, ~(x + 1)`
5. `x, ~x, x + 1, y`
6. `x, ~x, x + 1, y, z, ...`
7. `x, x + 1, ~(x + 1), y, z, ...`
8. `x, ~x, ~(x + 1), y, z, ...`
9. `x, ~x, x + 1, y, z, w, ...`

The first one represents a correct transmission, while 2-5 are transmissions with an error (modification), and 6-9 are transmissions with erasures. The recognition could have been implemented without an automaton, simply matching against the above cases in the order 1-9. We need to show, that for all the possible bytes `x`, the automaton will produce the correct output in all cases:

1. the automaton will follow the path 1->2->3->4->1, and produce `x` at its last transition.
2. the automaton will follow the path 1->2->5->7->1, and it will produce `~y` as its output, because the first character received as `x` is faulty, and the second matched character - received as `y` corresponds to $\sim x_{original}$, thus `~y` = $x_{original}$.
3. path followed will be 1->2->5->6->1, and `x` produced as the output in the last transition.
4. path followed: 1->2->3->6->1, output produced: `x` as above.
5. path followed: 1->2->3->4->8->9->3->..., and the output `x` will be produced in the 9->3 transition. In this case, we need to consume some bytes of the forthcoming quadruplet, in order to be able to distinguish between the cases 1, 5, and 9 (correct code, last byte lost, last byte modified). This becomes apparent only after we see the next three bytes, where we either see a pattern `y,~y,(y+1)`, in case of erasure, or `y, z, ~z`, in the case of modification, where `y` is the fourth byte in the original qadruplet. Note that in all three cases the output is the same, however, in order to remain synchronized in the next qadruplet, we ought to determine where it starts.
6. path followed: 1->2->5->1, output produced: `~x`, since the first byte of the qadruplet was lost, and thus the byte perceived as `x` was indeed $\sim x_{original}$.
7. path followed: 1->2->5->1, output produced: `x`, second byte was lost.
8. path followed: 1->2->3->6->2, output produced: `x`; here we had to read all four bytes in order to determine between the third byte lost, or modified, and thus we proceed directly to the state 2, if we found that the byte was lost and we already received byte from the next qadruplet.
9. path followed: 1->2->3->4->8->9->4->... output produced: `x`; same as in the case 5 above, we have to wait and see the incoming data in order to distinguish between erasure and modification.

Message length and checksum are also encoded with error-correction. When the

last byte of the message (according to the message length) is received, the recognition by the automaton is terminated. It is also terminated, when the end-packet terminating sequence is detected. If no transition can be followed in the current state, the data contain more errors than expected, the packet is ignored, and a new header packet expected. From the above, we see that each of the possible transmission error leads to the correct output, and the synchronization is preserved. Finally, we need to document that no more ambiguities occur, that is: in all states, when there are multiple matching transitions, we provide a static priority assignment over them, and this always leads to a correct error recognition and recovery. Looking through the states:

1. there is only one possible transition to state 2.
2. follows to the state 3, on `~x` (cases 1,4,5,8,9), otherwise to state 5 (cases 2,3,6,7). From case 6, we obtain the requirement that (`~x != x + 1`), i.e. (`0xFF - x != x + 1`), i.e. (`x != 0x7F`).
3. two possibilities: to state 4 (cases 1,5,9), otherwise to state 6 (cases 4,8), no ambiguities.
4. four possibilities: state 6 (case 3), state 7 (case 2), and two transitions to state 1 (cases, 6, and 7), no abiguities.
5. two possibilities: state 1 (cases 3,4), otherwise state 2 (case 8). Same output in both cases, only the synchronization is determined, no ambiguities.
6. only one possibility - state 1 (case 1).
7. single transition only (cases 5,9).
8. two options: state 4 (case 9), otherwise state 3 (case 5) (here we again receive `x != 0x7F`, and that `~(x + 1)` must not follow after `x`.

Provided that we encode the dangerous combinations with the ESC byte as described above, and given the assumption that the minimum distance between two errors (either modifications or erasures) is at least 4 bytes, the automaton should correctly restore the original message.

Other simple solutions might be considered. For instance, we could try to deal with the erasures and errors separately: Erasures could be detected using alternating bits (this would require transforming the original 8-bit byte-sequence into 7-bit data), and the errors could then take some sort of majority voting, or some standard efficient error-correcting code. This would most likely require breaking the bytes into bit sequences and bundling them to bytes to be transmitted in a different manner.

Conclusions

Error-correcting codes are useful tools for decreasing the amount of traffic in the communicating systems, where the correctness of data is important, and the communication channel is unreliable. A student project in our department required a reliable communication link between robots and the workstation, and attempted to use infrared signals translated and transmitted over the BT radio link. It has been encountered that the IR communication is unreliable in Lejos, and as a remedy, an error- and erasure- correcting protocol has been designed, implemented, tested and employed. The protocol is based on high data

redundance, but is relatively simple and can be decoded by an augmented finite-state automaton (which can also be suitable for embedded computing applications with limited CPU power, or for implementation directly in the hardware). We formally analyzed and proved correctness of the encoding on any possible failure in any two consecutive bytes.

References

[1] Petrovic P., Balogh R. (2006) Wireless Radio Communication with RCX, IDI/NTNU Technical Report 1/06.
[2] Friquin J.P.F. (2006) Designing and Implementing a Safety-Critical System for an Urban Search and Rescue Robot. Master thesis (in progress), IDI, NTNU.
[3] McEliece R.J. (1977) The Theory of Information and Coding. Addison Wesley.
[4] Lin S., Costello D.J. (2004) Error Control Coding, Second Edition. Prentice Hall.
[5] Morelos-Zaragoza R.H. (2002) The Art of Error Correcting Coding. John Wiley & Sons.
[6] Rizzo L. (1997) Effective erasure codes for reliable computer communication protocols. ACM Computer Communication Review, vol.27, no.2, pp.24-36.
[7] BT2IR SourceCode: http://www.idi.ntnu.no/grupper/ai/eval/
[8] BrickOS home: http://brickos.sourceforge.net/
[9] Lejos home: http://lejos.sourceforge.net/