Analysis of the NXT Bluetooth-Communication Protocol

By Sivan Toledo

September 2006

The NXT supports Bluetooth communication between a program running on the NXT and a program running on some other Bluetooth device. The other device can be another NXT, a desktop or laptop computer (which I'll refer to here as a PC), a mobile phone, etc. This article investigates the performance and reliability of this Bluetooth communication.

The NXT can also receive various commands via Bluetooth: to activate motors, to read sensor input, to run programs, to manage files, and so on. The performance of this aspect of the Bluetooth communication is not the focus of this article; you can deduce some things about it from this article, but it mainly focuses on program-to-program communication.

This article can be used in three ways. If you are designing NXT programs that communicate via Bluetooth, this article will tell you (perhaps with too many details :-) how to exchange messages reliably and/or quickly. You may think that if a program on one NXT sends a message and a program on another tries to receive a message than the message will indeed be received (after all, if your program tells a motor to start it does start), but this is not always the case.

If you are designing software for non-NXT Bluetooth-enabled devices and you want this software to communicate with NXT bricks, then this article will help you write software that communicates correctly with the NXT.

Finally, if you are interested in communication protocols, then this article has two educational uses. First, it can be used to discuss reliability in communication protocols: why we might want a protocol to be unreliable and how to design a reliable protocol on top of an unreliable one. Second, it can be used to study (and perhaps criticize or improve) the experimental methodology that should be used to measure the reliability and performance of systems.

The Protocol

The NXT program-to-program communication protocol classifies the two connected computers into a master and a slave. They behave differently. The master is always the side that created the Bluetooth connection. If you create the connection from

the Bluetooth menu of the NXT, that NXT is the master. If the connection was created by a PC or by some other NXT, the NXT that simply accepted the connection request is a slave.

On a PC, the easiest way to connect to a slave NXT is using Fantom, a driver that is supplied and installed together with the NXT-G programming environment that comes with the NXT. If a NXT connects to a PC and the PC is the slave in the connection, the connection looks like a working serial port connecting the PC with the NXT.

The principle behind the master-slave relationship is simple: only the master is supposed to initiate communication. A message from the master may carry program-supplied data, or it may ask the slave for data that the program on the slave sent to the master. When the master sends data to the slave, it may request an acknowledgement or it may request that the slave accepts the data without acknowledging it. When the master requests data from the slave, the response contains the data if data is available, or an error code if no data is available or if no program is running on the slave.

The rationale behind this protocol appears to be that it allows both sides to always know who is supposed to be sending and who is supposed to be receiving. When the connection is idle, both sides know that the next event will be a message from the master. Once a message from the master is sent, it specifies whether a reply is required, so again both sides know whether the master or the slave will transmit next. After the request is sent (if a request was required), the connection goes idle again and the process repeats. The Bluetooth ratio on the NXT switches slowly from receive mode to transmit mode, and this protocol eliminates mode transitions that might have been caused by uncertainty as to who transmits next.

When a program sends or receives a message, it specifies a mailbox, which is a number between 0 and 9 (represented as 1 to 10 in NXT-G programs). For the slave, this builds 10 virtual channel on top of one master-slave connection. The master's use of mailboxes is a bit more complex. A NXT master can be connected to up to 3 slaves. When the master sends a message, the program specify to which slave and to which mailbox to send the message. But when the master receives a message, it only specifies a mailbox. The mailbox may contain a message from any of the 3 slaves. The program on the master cannot know from which slave the message was received (unless, of course, the contents of the message provides this information).

The NXT's firmware support for program-to-program communication consists of 4 system calls. Two system calls are supposed to be used only on the master, `NXTCommBTCheckStatus` and `NXTCommBTWrite`. The first checks whether the Bluetooth radio is busy. If it is, the program is supposed to wait until the radio no longer reports that it is busy. The second system call, `NXTCommBTWrite`, delivers a

message to be sent to the radio. It should only be called when the radio is not busy. This is how the master sends program data to the slave: by delivering messages directly to the ratio. The firmware does not manage a queue of messages to be sent by the master. To receive a message, the master calls a third system call, `NXTMessageRead`, and tells it from which mailbox it wishes to read. The mailbox should be empty, because the slave is not supposed to initiate communication even if it has messages to deliver. So what `NXTMessageRead` does is to send a request to the slave to send a message to the particular mailbox on the master. There are three possible outcomes to this request. If the program on the slave already sent a message to the mailbox, then this message is waiting on the slave, and the firmware of the slave will send it back in the reply message. If no message is waiting on the slave, or no program is running on the slave at all, it will send back an error code indicating why it cannot deliver a message. If the slave does not reply at all, the system call will time out and will return without delivering a message to the master's program.

The program on the slave uses two system calls to communicate with the master. When the program wants to receive a message from the master, it also calls `NXTMessageRead`. On the slave, this system call behaves differently than on the master. It simply checks the specified local mailbox and returns whether or not the mailbox contains a message. It does not poll the master for messages, because (a) the slave is not supposed to initiate communication, and (b) the master sends messages immediately anyway, so it is not possible that a message is simply waiting on the master. To send a message to the master, the slave calls `NXTMessageWrite`. This system call is used only by slaves. It adds the message to a queue belonging to the specified mailbox. The message will wait there until the master retrieves it by polling the slave for that mailbox.

Reliability Consideration: The Underlying Bluetooth Protocol and the Mailbox Queues

The NXT's Bluetooth communication uses a Bluetooth application protocol called the Serial-Port Profile (SPP). This application-level protocol is implemented on top of a low-level protocol called RFCOMM. RFCOMM is a reliable transport protocol. What this means is that data delivered to RFCOMM on one side is always delivered to the other side of the Bluetooth communication channel, unless the connection is broken. RFCOMM is not allowed to loose data.

But this does not mean that the NXT Bluetooth protocol does not loose messages: it does, because the mailbox queues are short. Each mailbox queue can contain up to 5 messages. If a message must be pushed into a full queue, the oldest message in the queue is deleted. In principle, only the slave needs queues. The master sends

messages immediately, and it should only receive messages when it polls the slave because the master's program asked for a message.

Because old messages simply get deleted from full queues, messages in the protocol are not always delivered. Messages from the master to the slave are held in a queue on the slave. If the program running on the slave does not poll frequently enough for incoming messages but the master keeps sending messages, then the queue will overflow and old messages from the master will be lost. Similarly, if the program running on the slave keeps sending messages but the master is not polling the slave fast enough, old but undelivered messages get deleted from the slave's outgoing mailboxes. Worse, if the slave's program sends more than 5 messages in a row without pausing, there is essentially no way for the master to retrieve them quickly enough; the slave is storing messages into buffers in its local RAM, but the master must retrieve them over Bluetooth. In such situation, all but the last 5 messages to be sent from the slave are usually lost.

There is yet another reason for messages to be lost. When there is no program running on the slave, the slave's firmware ignores Bluetooth-communication messages. It replies with an error code if a reply is requested, but it does not queue incoming messages and it does not send back outgoing messages from a program that is no longer running. Therefore, if the master sends messages before the slave's program starts, these messages will be lost. Also, if the slave sends a message (or a few) and ends it's program, these last messages are typically lost, because the queues are cleared before the master polls for them.

The firmware developers built an unreliable protocol on top of a reliable transport protocol (SPP over RFCOMM). This is a little strange. In many cases, reliable high-level protocols are built on top of unreliable low-level protocols. For example, TCP is a reliable protocol that is built on top of the unreliable IP protocol. But there is a reason for the design decision made by the firmware developers. With limited memory to buffer messages, the only way to design a reliable protocol is to block the sender when the queue is full. If the receiver is not pulling messages from the queue quickly enough and the sender still wants to send messages, you can either drop messages or block the sender (make it wait until the queue empties), but you cannot avoid both. The developers of the NXT decided not to block the sender.

The NXT-G Bluetooth communication blocks also adhere to the non-blocking philosophy. They always complete their execution within a fixed amount of time (less than a second, usually a lot less). This means that the send block on the master does not always wait for the radio to be available; if the radio reports that it is busy for long enough, the block returns without sending the message, even though it could have waited for the ratio to become available. The send block on the slave always delivers a message to the mailbox queue and returns, even though this may overflow the mailbox queue and cause old messages to be deleted. (The firmware does not

provide a system call to determine how full a queue is.) The receive blocks on both the master and the slave return without delivering a message to the program if they cannot deliver one quickly.

Ways to Avoid Queue Overflows

There are several ways to avoid queue overflows and message loss. Some of them entail some inefficiency, not all are possible on the NXT side (but they are on the PC side), and not all are possible in a NXT-G program.

The simplest way to avoid losing messages is to ensure that the queues are empty before sending messages. Suppose that the master and slave programs both alternate between sending and receiving messages. Each one of them sends one message, then waits for a message to arrive for the other side, and then they send another, and so on. This ensures that every queue never contains more than one message, so queues cannot overflow. Of course, the programs could send up to five messages before receiving and the queues would still not overflow. To avoid message loss because the slave does not run a program, the sequence should start with a message sent from the slave (the master starts by receiving), and it should end with a message received by the slave.

This technique is essentially a way to build a reliable protocol on top of an unreliable one.

Alternating between sending and receiving reduces the performance that you can get from the protocol. The performance loss is caused both by the need to repeatedly transition from receive mode to send mode, and by messages that might carry no useful data (beyond the implicit indication that the queue is empty). The experiments described below indicate that you should expect each iteration of this protocol to take a little over 50 ms.

Let us see what can be done to increase the reliability of message exchange without such a large performance penalty.

One way to avoid at least some queue overflows is to have bigger queues. On the NXT, this is not possible, because the firmware was designed to use 5-message queues (because RAM is quite limited). But if a computer with more memory, such as a PC, is communicating with a NXT, we can usually use very long queues. This can avoid most of the message loss in a channel in which the NXT is the master. This technique is useless when the NXT is the slave, because queue overflows occur at the slave.

This technique is not always useful; there are positive aspects to message loss. The contents of a message may be useful for only a certain amount of time. For example,

if messages from the PC to the NXT carry motor-actuation commands, it may be better to not deliver old messages than to deliver them. If the NXT did not receive and therefore did not respond to an old command, it may be better to issue a new command that is more approriate to the current situation than the deliver and execute the old command. So if queues on the PC side are long, it may be useful to tag messages with a deadline or some other data that will indicate whether to deliver or to drop them when their turn arrives.

Another technique, on a NXT that serves as a master, is to wait until the radio is ready before sending a message. This is not possible in NXT-G, since the send block in NXT-G may time out before the radio is ready. It appears that the timeout was chosen to be large enough to avoid most send-failures when the connection is functional, but not too large to slow down programs. As we shall see below, the timeout value is usually large enough, but not always; the send block on a master may fail to send a small fraction of the messages due to this timing out.

When a PC master is sending messages to a NXT slave, it is possible to avoid overflowing the slave's incoming queue using an undocumented feature of the protocol. When the master sends a message, it can specify in its header whether it wants the slave to send a reply with a success/error code. If the master does not requests a reply, the slave's firmware queues the incoming message, deleting the oldest message in the queue if it is full. It turns out, however, that if the master requests a reply, the slave only sends the reply when it is able to queue the message without overflowing the queue. This is an undocumented behavior of the firmware, but my experiments indicate that the firmware behaves as described (at least firmware version 1.0.3).

Using this feature entails an inefficiency. The NXT's Bluetooth radio takes a long time to transition from receive mode to transmit mode, and sending an acknowledgement requires such a transition. The performance measurements described below show that this is a little faster than send/receive alternatation at the program level, but not by much (47 ms versus 54 ms).

It is easy to use this feature from a PC master. If the PC program uses the Fantom driver, it simply requests a response. The call to the Fantom driver only returns when the request arrives. I am not sure whether it is possible to use this feature from a NXT master. The only indicator that is available to a NXT master program about the status of a sent message is the success/error code of the `NXTCommBTCheckStatus`. To be consistent with the behavior of the Fantom driver, it should indicate that the radio is busy until the radio received the reply from the slave. But I have not verified whether it behaves like this or not. In any case, it is not possible to use this feature in a NXT-G program, because if the slave delays the response, the send block will time out anyway. Also, one cannot control whether replies are requested or not when a NXT-G sends a message (but it is possible to control this at the system-call level).

Performance

Several factors affect the performance of the NXT's Bluetooth communication. The most significant factor is the large amount of time the NXT's radio needs to transition from receive mode to transmit mode, about 30ms to according to the documentation (LEGO MINDSTORMS NXT Communication Protocol document, version 1.00, 2006, page 22).

The connection between the NXT's main CPU and the Bluetooth module can transfer at most 58 kilobytes per second (460 kbits/s). This channel carries both data and various commands. This is more limiting than the bandwidth of the radio interface, which ranges from 723 kbits/s (for Bluetooth 1.1 and 1.2) to more than 2 Mbits/s (for Bluetooth 2.0 with Enhanced Data Rate; the radio of the NXT support 2.0+EDR).

Another consideration is the fact that messages also carry some NXT-protocol data, not only program data. Messages from the master carry 7 additional bytes per message. Protocol messages in which the master polls the slave for data are 7 bytes long and carry not program data. Replies from the slave carrying program data are always 66 bytes long, no matter what the actual length of the message is. This means that short messages from the slave entail a particularly large amount of overhead.

Experimental Setup

The performance and reliability measurements reported below were carried out using a NXT running firmware 1.0.3 and a PC running Windows. The PC is an IBM R50p Thinkpad with a built-in Cambridge Silicon Radio Bluetooth radio. The Radio on the PC is an IBM Integrated Bluetooth II manufactured by Cambridge Silicon Radio, supporting Bluetooth 1.1, and driven by a WIDCOMM device driver version 1.4.2.8.

The experiments were conducted in an environment in which no other Bluetooth radios could be detected.

Measurements with the NXT as a Master

In the first experiment I sent 1000 packets of text from the NXT to the PC. I measured both short packets (4 bytes of text, which result after you add the NXT protocol's header and trailing zero byte in 4+7=11 byte messages over the serial connection) and longer messages of 25 or 47 bytes of text. The messages themselves are static strings and the sending loop does nothing but call the NXT-G Bluetooth-send block (or an equivalent NBC subroutine).

I used both NXT-G and NBC programs to carry out the NXT side of the interaction. The programs are the same in terms of functionality.

Here are the results:

	Text-Length of Message	Time per Message	Useful Bandwidth
NXT-G	4 bytes	3.0 ms	1.32 Kbyte/s
	25 bytes	3.6 ms	6.75 Kbyte/s
	47 bytes (0-2 dropped messages)	4.0 ms	11 Kbyte/s
NBC	4 bytes	3.0 ms	1.32 Kbyte/s
	25 bytes	**3.0 ms**	**8.2 Kbyte/s**
	47 bytes	4.0 ms	11 Kbyte/s

There is not a big difference in performance between NXT-G and NBC. There is some difference in message of medium length (25 bytes). The more interesting difference is that the NXT-G program dropped a few messages when messages were long (in my experiments, between 0 and 2, but dropping was not rare). The NBC program waits until the radio reports that it is not busy (using a firmware system call). The NXT-G Bluetooth-send block can time out before a message is sent when the radio is overloaded by long messages.

In the next experiment, the NXT asks the PC for 1000 messages and then receives them in a loop. Within this loop, the program executes another loop. In this inner loop, the NXT tries to receive a message. The inner loop ends when the NXT indeed receives a message. The NXT program is written in NXT-G. The replies from the PC are all 4 bytes long.

This program received all the 1000 messages in 38 seconds (38 milliseconds per received message). This is consistent with the LEGO-published documentation, which states that the Bluetooth chip in the NXT requires about 30ms to transition from receive mode to transmit mode.

The third experiment is a ping-pong experiment. The NXT sends 1000 messages to the PC (9 bytes each). After each message is sent, the NXT program enters an inner loop in which it waits for one 4-byte message from the PC. This NXT program is also written in NXT-G.

The whole process took 54.6 seconds, or 54.6 milliseconds per request-reply cycle. This is a little harder to understand. In each one of the 1000 iterations of the outer loop, the NXT master sends two messages, one from the send block and another from the receive block (polling the PC for a message), and receives one message from the PC. The messages from the master do not request an acknowledgement. Therefore, there is only one transition from receive to transmit per iteration. It is not completely clear to me why each iteration is taking 54 ms rather than about 42 (38 to

send and receive once, as in the previous experiment, plus 3-4 more to send the other message to the PC from the send block).

The NXT as a Slave

When the NXT is the slave in the connection, messages are much more likely to be dropped. As mentioned above, if the program on the NXT slave sends more than 5 messages in a row, all but the last 5 are usually dropped. Waiting 50 ms after sending each message avoids this loss, as long as the Master is constantly polling the slave (and not sending messages of its own). Also, if the communication ends with a message from the slave, it is necessary to wait after sending it, because the slave will not deliver the message once it is no longer running a program.

The experiment in which the NXT sends 1000 messages ended, predictably, in almost all the messages being lost. In several repetitions, fewer than 40 out of the 1000 messages were actually delivered to the PC. All the other ones were deleted from the mailbox queue when it overflowed. Waiting 50 ms after every message is sent eliminates the lost messages, but the running time per message obviously rises to a little over 50ms.

In the NXT experiment the NXT slave receives 1000 messages in a row. I ran the experiment in two different ways. When the PC Master sent the messages without asking the slave to acknowledge them, between 800 and 820 messages were actually delivered, at a running time of 4.6-4.9 ms per message. This is a little slower than the performance in the other direction (NXT as a master), but still much faster than all the other modes of communication. This way of sending messages from the master is exactly how the NXT-G send block works: without requesting an acknowledgement.

When the PC master requested an acknowledgement for each sent message, performance dropped to about 47 ms per message, but no messages were lost.

In the ping-pong experiment each iteration took 93 seconds.

Using Multiple Mailboxes (Concurrently or Sequentially)

Using multiple mailboxes can be useful from the software-engineering point of view. Data of different types (text strings, numbers, boolean values) can be sent to different mailboxes, allowing the receiver to determine the data type from the inbound mailbox number. Data with different meanings (e.g., sensor data from two different sensors) can be sent to different mailboxes. But achieving good performance with multiple mailboxes can be tricky.

For the master, there is no difference between sending multiple messages to a single mailbox or to multiple mailbboxes. But there is a big difference between receiving messages into one mailbox or into several. When the master is trying to read multiple mailboxes, it must poll the slave for each mailbox separately. Each polling attempt will take at least 30 ms on both sides. Polling for multiple empty mailboxes wastes a lot more radio time than polling for a single mailbox.

If you decide to receive messages from multiple mailboxes on a master, you need to decide whether to poll for messages concurrently from multiple threads or to poll sequentially for each mailbox (and repeat). On a master, it is not a good idea to use the radio from multiple threads unless all the threads are sending messages. The NXT's firmware uses the radio sequentially, so only one thread can use it at a time. If one or more threads are polling for messages, the radio will be busy for fairly long amounts of time (at least 30 ms per polling attempt). This will cause long delays for the other threads, whether they are sending or receiving. Furthermore, the NXT's scheduling of thread's access to the radio is not always fair. So one thread may perform three unsuccessful polling attempts while the others (who might actually have messages waiting on the slave) are starved. This can cause long delays in delivering messages. The long delays can cause message loss on the slave, unless your application-level protocol is reliable (but if it is reliable, it will be very slow in such cases).

So we better poll sequentially. But if we poll sequentially, there is not much point in using multiple mailboxes. Suppose that we want to use 5 different "channels". We could use 5 mailboxes, but could use instead an alternating protocol that works as follows. The master sends an explicit request message. Once received by the slave's program, the slave responds with exactly 5 messages, all to the same mailbox. But the first would contain the reply to the first "channel", the second the reply to the second "channel", and so on. If the slave has no data for a certain "channel", it sends an agreed-upon invalid content to indicate the lack of data. This is a reliable protocol, because the queues would never overflow. It is fair, in the sense that all the "channels" get equal use of the radio, and it is not much slower than polling for 5 different mailboxes.

Conclusions

The overall conclusions are

1. The NXT's Bluetooth communication protocol is unreliable (by design). It's easy to loose messages. If your system requires reliable communication, make sure that you use the NXT's protocol in a way that ensures reliability.
2. The performance differences between reliable program-level alternating protocols and most of the other reliable ways (but not all the other ways) to

exchange messages are not large, so unless performance is critical, alternating protocols are best. They take 50-100 ms per iteration, depending on the devices used (two NXT's vs. a NXT and a PC).

3. To avoid message loss in alternating protocols due to late startup or early termination of the slave, start the alternation with a message from the slave and end it with a message from the master.
4. If you want higher performance than you can achieve with an alternating protocol, you really have just two options. If you need to send data quickly from the NXT, configure it as a master and send data to a device that will accept incoming messages without dropping them (e.g., a PC). If the messages are short, this works reliably even with the timeouts built into the NXT-G send block. If messages are long, you should use a another programming environment that does not time out when the radio is busy.
5. If you need to send data quickly into the NXT, configure it as a slave and send the data without requesting acknowledgement (this is also how the NXT's firmware sends data from a slave). You should expect some message loss (in my experiments, about 18-20%), but the data will be delivered in less than 5 ms per message. There is apparently no way to send data as quickly to the NXT in a reliable way.
6. Receiving messages into multiple mailboxes on a master usually has a negative impact on performance without a significant advantage in software engineering. It is also hard to design reliable program-level protocols that use multiple mailboxes.

One of two relatively small improvements in the NXT's software would improve the usability of its Bluetooth protocol. One improvement would be to add string-splitting and text-to-number blocks to NXT-G. These two are easily implemented using the existing instruction-set of the underlying virtual machine, and they are symmetric to two existing blocks, the string-concatenation block and the number-to-text block. These new blocks would allow programs to send multiple "channels" of data to a single mailbox, distinguishing the channel by the contents of the message. For example, to send data from both a light and a sound sensor, the program would send messages such as `sound 87` or `light 13` (easy to format using existing blocks). The receiving program would use the new blocks split the messages into words and to convert the second word into a number.

Another way to achieve the same goal would be to allow wildcards in the `messageread` direct command, which the master uses to poll for messages on the slave. Instead of specifying a mailbox between 1 and 10, the master would use a number, say -1, that would indicate that it is willing to receive a reply containing a message from any non-empty queue. If the slave also runs a firmware that supports this wildcard, it would reply appropriately. If the slave runs older firmware (like today's version 1.0.3) it would send back error code `0xEE` (invalid mailbox number). This would indicate to the master's firmware that it needs to fall back to compatibility

mode and poll for specific mailboxes. This is a more complex solution whose main advantage over the first is that it would allow user to use the mailbox abstraction with a smaller (or no) performance penalty.

A more radical (and perhaps not backward compatible) change would be to tag outgoing messages with importance (or expiration times). The protocol would need to change such that important messages are never dropped. This would necessarily imply that the sender would sometimes be blocked, perhaps for a long period of time.