How Bitcoin Satellite Works

April 8, 2021

Contents

1 Bitcoin Satellite

Bitcoin Satellite is an application based on Bitcoin FIBRE and Bitcoin Core. It is the application used to transmit and receive blocks over the Blockstream Satellite network. The transmitter (Tx) nodes maintained by Blockstream use this application to send blocks over the satellite links. The receiver (Rx) nodes run by users worldwide, in turn, rely on this application in conjunction with specialized receiver hardware to receive the blocks from space.

As a fork of Bitcoin Core, Bitcoin Satellite uses the same applications: `bitcoind` or `bitcoin-qt`. The difference is that it introduces two new options: `-udpmulticasttx` and `-udpmulticast`. These options implement the transmission and reception of data over one-way multicast-addressed UDP streams. Such UDP multicast streams can carry blockchain data with unique mechanisms that are suitable for satellite transmissions.

You can install Bitcoin Satellite directly from Blockstream Satellite's command-line interface (blocksat-cli). Please refer to the project's documentation for further instructions.

In this guide, we thoroughly explain the mechanisms introduced by options `-udpmulticasttx` and `-udpmulticast`. More generally, we explain how Bitcoin Satellite works and how you can use it effectively.

1.1 Fundamental Concepts

To start, we shall explain some of the fundamental concepts. If you understand them now, you will have an easier time understanding the main features of Bitcoin Satellite.

1.1.1 Forward Error Correction (FEC)

The main goal of a forward error correction (FEC) mechanism is to transport data reliably over a communication channel that can corrupt the data (i.e., a lossy channel). An FEC mechanism increases the chances of recovering a data object despite the occurrence of data loss. For example, when receiving data over a satellite link, the receiver will often miss parts of the data. In this context, an FEC scheme comes handy.

An *FEC encoder* adds redundant information to a data object. On the receiving end, in turn, an *FEC decoder* tries to recover the original data based on the received information (including the added redundancy). If the excess (redundant) information is sufficiently long, the receiver can recover the data even if there are lots of errors or missing pieces.

In general, an FEC mechanism works as follows:

That is, the sender applies the FEC encoding, and the receiving end executes the decoding step.

1.1.2 Erasure FEC Coding

A particular type of FEC scheme consists of the so-called **erasure codes**. What's different about them is that they focus on data received with missing pieces

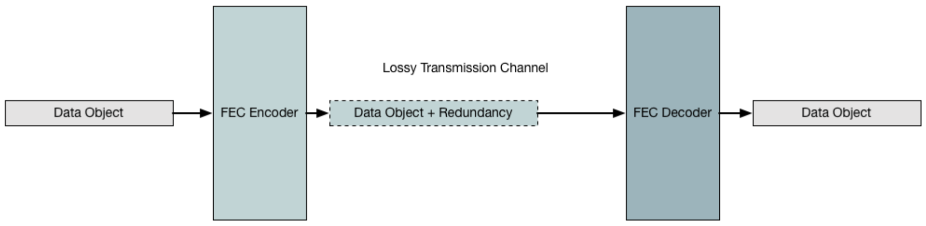

Figure 1: General operation of FEC encoders and decoders.

rather than data acquired with errors. For example, suppose the original data can be split into several chunks of equal size, as follows:

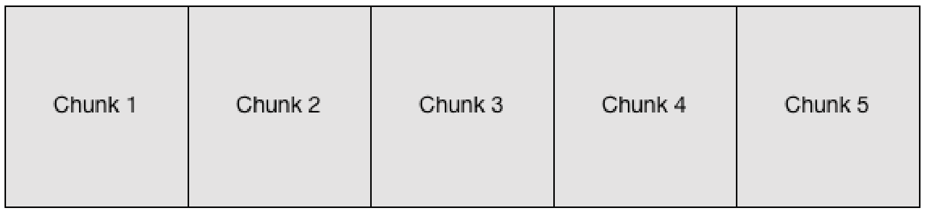

Figure 2: Example of data object split into equal-sized chunks.

Next, suppose that all chunks are sent over a satellite link, but the receiver only gets some of them:

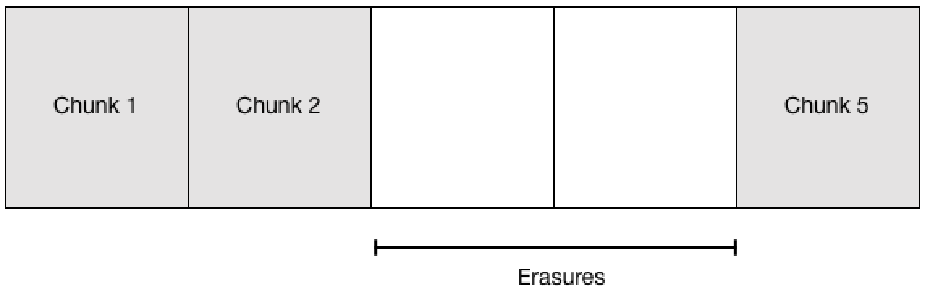

Figure 3: Erasures: missing chunks of data.

That is, in this example, the receiver misses chunks 3 and 4.

The primary goal of an **erasure code** FEC mechanism is to fill the missing pieces, which are called **erasures**.

The way the mechanism works is again by using redundancy. Instead of transmitting only the original chunks of the data object, the transmitter also sends extra (redundant) pieces. With that, the receiving end can tolerate the loss of some chunks. As long as the receiver gets enough of them, it can still recover the original data. Thus, the FEC mechanism becomes as in the following illustration:

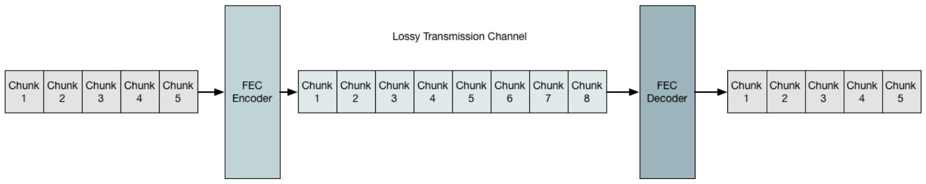

Figure 4: General operation of an erasure coding FEC mechanism.

You can think that one way to accomplish the same thing would be to transmit the original data chunks repeatedly. For example, instead of sending each chunk only once, one could send each chunk twice. With that, the receiver would have higher chances of collecting all the data pieces. Nevertheless, this is an inefficient approach. An erasure FEC mechanism can recover the data much more efficiently than a simple data repetition strategy.

The primary efficiency mechanism is the ability to recover the original data based on **any** set of distinct chunks. We illustrated above an example where chunks 3 and 4 out of five fragments were missing on reception. However, suppose that the receiver, now featuring an FEC mechanism, does lose chunks 3 and 4 but receives two extra (redundant) chunks, like so:

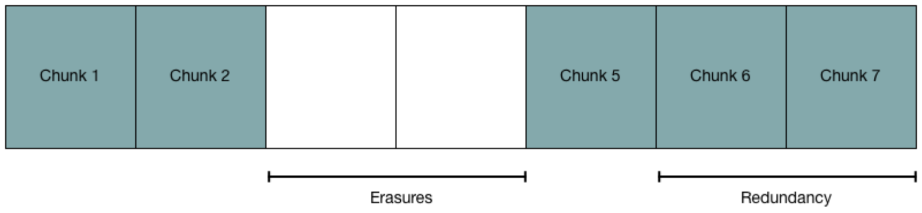

Figure 5: Reception of redundant FEC chunks.

In this case, a well designed FEC mechanism can recover the original data based on the received set of five distinct chunks (1, 2, 5, 6, and 7), even though chunks 6 and 7 (redundant) are not exactly identical to the missing pieces (3 and 4). If there are two missing chunks, the receiver can complete the FEC decoding as soon as it acquires two extra distinct chunks. In contrast, a mechanism based on repetition needs to receive the exact missing pieces. In this case, it would wait until the repetitions of chunks 3 and 4 come, which is a far more inefficient approach.

The erasure coding approach is the one adopted on Bitcoin Satellite. With the specific transmission protocol used by the Blockstream Satellite network (the DVB-S2 standard), it turns out that the receivers only output correctly-recovered chunks of data. If a specific data fragment is wrongly received, an *integrity* *check* identifies the problem and drops it. As a result, the receivers may only miss some chunks (the ones that are dropped) but never receive chunks with errors. Hence, an erasure coding FEC scheme is perfectly suitable to overcome the chunk losses.

1.1.3 UDP and Multicast Addressing

UDP is a connectionless protocol for transmitting data over a network. Once a UDP sender sends a message, it does not wait for a reply or an acknowledgment from the recipient. Furthermore, the UDP sender does not retransmit the data if the UDP recipient misses it (it cannot tell whether the recipient missed the message). Hence, UDP is a lightweight protocol suitable for one-way transmissions. It is opposed to the widely used TCP, which includes connections, retransmissions, and other features, but requires two-way communication. UDP is used in Bitcoin Satellite primarily due to its one-way nature, as satellite receiver nodes can only receive data but not transmit.

The concept of multicast addressing refers to the destination of the UDP messages. A point-to-point UDP transmission scheme is called unicast-addressed. It consists of one sender node that posts a message to a specific recipient. In contrast, with multicast addressing, the communication becomes point-to-multipoint. A multicast transmitter node (think a satellite transmitter) sends a message for many recipients simultaneously (the satellite receivers). A multicast receiver, in turn, consists of a node or application interested in a particular multicast destination address. As discussed later, the Bitcoin Satellite application runs such a multicast listener.

1.2 Satellite Transmissions

Next, we discuss the Blockstream Satellite transmissions that originate from the Bitcoin Satellite application. We shall distinguish them in terms of *streams*, which refer to independent sequences of multicast-addressed UDP packets.

The Blockstream Satellite network continuously broadcasts four independent streams:

- **Stream 1**: Newly mined blocks arising in real-time on the peer-to-peer (p2p) Bitcoin network.
- **Stream 2**: The past 24h of blocks (past 144 blocks on average).
- **Stream 3**: The past 1h of blocks (past 6 blocks on average).
- **Stream 4**: The full blockchain.

Stream 1 allows bitcoin satellite receiver (Rx) nodes to stay in sync with the blockchain in real-time. As soon as a new block propagates over the p2p network, it is also broadcast over the satellite network.

Stream 2 and **Stream 3** allow Rx nodes to recover from recent losses or temporary reception outages. A satellite receiver can experience complete interruptions or significant reception failures due to weather conditions (primarily under heavy rain). If such a failure leads to missing a block when it is sent over Stream 1, the Rx node can still download the block later when repeated over Stream 2 or 3.

Stream 4 continuously propagates the entire blockchain. One could set up a fresh new Bitcoin node attached to a satellite receiver and still achieve the full sync (i.e., the initial block download) via satellite only.

The satellite network currently allocates the satellite link capacity as follows: - Stream 1 uses the total capacity whenever there is a new block to be transmitted. It is also the stream with the highest priority on transmissions. - Streams 2 and 3 operate with approximately 70 kbps each. For Stream 2, this bitrate is enough to cycle over 144 blocks at least three times during 24h. For Stream 3, likewise, this bitrate is enough to cycle through 6 blocks at least three times throughout an hour. - Stream 4 uses the remaining capacity, which is approximately 870 kbps on average. As a result, the complete sync can be accomplished in around 36 days when receiving from a single satellite or roughly 18 days with dual-satellite reception.

Thus, conceptually, the multiplexing of streams looks as follows:

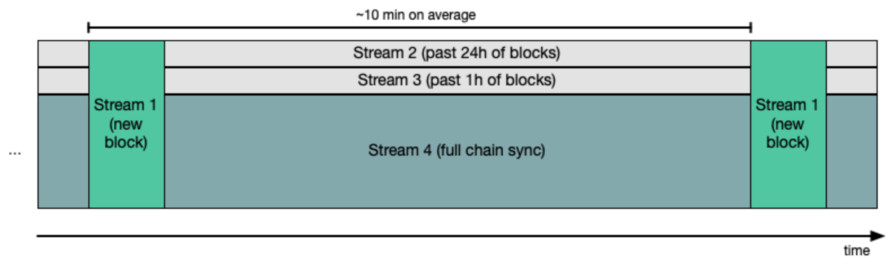
~10 min on average
Stream 2 (past 24h of blocks)
Stream 3 (past 1h of blocks)
Stream 1 (new block)
Stream 4 (full chain sync)
Stream 1 (new block)
...
...
time

Figure 6: Time multiplexing of Blockstream Satellite streams.

Note that Stream 1 only runs when there is a new block to send and then sleeps until the next block.

1.3 Satellite Reception

A Blockstream Satellite receiver receives the multicast-addressed UDP packets sent over satellite. Upon reception, these messages are conveyed over the local network to the interested multicast listeners.

A Bitcoin Satellite Rx node runs `bitcoind` with option `-udpmulticast` on its `bitcoin.conf` file. This option leads to `bitcoind` joining the multicast listeners interested in the particular address (`239.0.0.2:4434`) to which the Blockstream Satellite transmitters send the UDP packets.

1.4 How Bitcoin Satellite Achieves Reliable Transport of Blocks

Bitcoin Satellite aims at transporting Bitcoin blocks reliably over the lossy satellite channels. In other words, it tries to maximize the chances of carrying blocks successfully to receivers worldwide, despite the random losses that each receiver can experience.

The way it achieves reliable transport is based on the FEC schemes mentioned earlier, particularly the erasure coding approach. Each block is split into multiple chunks of data, which we call *FEC chunks*. Then, the FEC chunks from distinct blocks are alternated over time to maximize the outage interval that a receiver can tolerate. This process is thoroughly explained next.

1.4.1 FEC Chunk Generation and Decoding

As mentioned earlier, each block transmitted via the UDP multicast service is first divided into multiple chunks of data, each containing 1152 bytes. Subsequently, the Bitcoin Satellite sender node post-processes all chunks of a block through the so-called *FEC encoder*. The encoder takes `x` original fragments and produces `y` completely different (FEC-encoded) chunks, where `y > x`. In other words, it encodes (modifies) the data to facilitate decoding and adds extra (redundant) chunks (given that `y > x`). For example, when processing a block that originally occupies 1000 fragments, the transmitter node can send 1110 FEC chunks, among which 110 pieces are redundancy.

Each FEC chunk is sent on an independent multicast-addressed UDP datagram. The UDP multicast listeners (satellite receivers), in turn, collect the chunks and track whether a sufficient number of them has been received for each block. Once enough FEC chunks have been acquired, the receiver decodes the block.

In most cases, the Rx node can decode a block by receiving any combination of `x` distinct chunks, `x` being the number of fragments derived from the original block before the FEC encoding. Hence, the receiver needs to track how many different chunks it has acquired for each block coming via satellite. Once it has `x` unique chunks for a given block, it starts trying to decode the FEC object.

The actual number of chunks sent by the transmitter nodes for each block is `y` and `y > x`. Hence, there are `y - x` excess chunks. This surplus corresponds to the number of fragments that the Rx node can miss while preserving its ability to recover the block. In the example where the Tx node sends `y=1110` chunks for a block that originally has `x=1000`, an Rx node can miss up to `110` chunks and still recover it.

So the two key elements of the decoding process are the redundancy and the efficient data recovery mechanism provided by the FEC scheme. The redundancy element comes from the transmitter sending `y` FEC chunks for each block of `x` original chunks, with `y > x`. The efficient data recovery comes from the FEC scheme's ability to recover the original data from any set of `x` distinct chunks on the receiving end. It implies that the Rx node can lose any set of `y - x` chunks safely most of the time.

1.4.2 FEC Chunk Encapsulation and Tracking

The next question is how the Rx node can track the incoming FEC chunks. The Bitcoin Satellite implementation uses the encapsulation protocol from Bitcoin FIBRE with minor modifications. This protocol adds enough information into the UDP packets to allow receivers to track the incoming FEC chunks.

In essence, each UDP datagram contains metadata in addition to the actual FEC chunk. The essential metadata fields are the following:

- Hash prefix
- Original data object length
- Chunk ID

The *hash prefix* is a 64-bit prefix of the block hash. At first, the receivers rely on this information to organize which chunk belongs to each block. The *original data object length* is the information that allows receivers to figure out how many chunks are necessary for each block. That is, it gives enough information to compute `x`, the number of fragments derived from the original block (before FEC encoding). Lastly, the *chunk ID* is the chunk identification number, which allows for tracking of the unique (distinct) FEC chunks received for each FEC-encoded block.

1.4.3 Interleaved Chunk Transmissions

So we already know that each Bitcoin block is split into multiple FEC chunks and that each chunk is sent on an independent UDP packet with some metadata fields. The next important aspect is how we alternate between chunks of different blocks. The strategy differs among the four streams sent over the satellite network.

Recall that Streams 2 and 3 refer retransmit the past 24h and 1h of blocks, respectively. Recall also that Stream 4 refers to the full-blockchain transmission loop. The Tx node interleaves (alternates in time) the chunks of several consecutive blocks in all of these streams. First, the Tx node groups several successive blocks in a window of blocks. Then, it transmits a single chunk out of each block per transmission turn.

For example, consider the hypothetical window of blocks below:

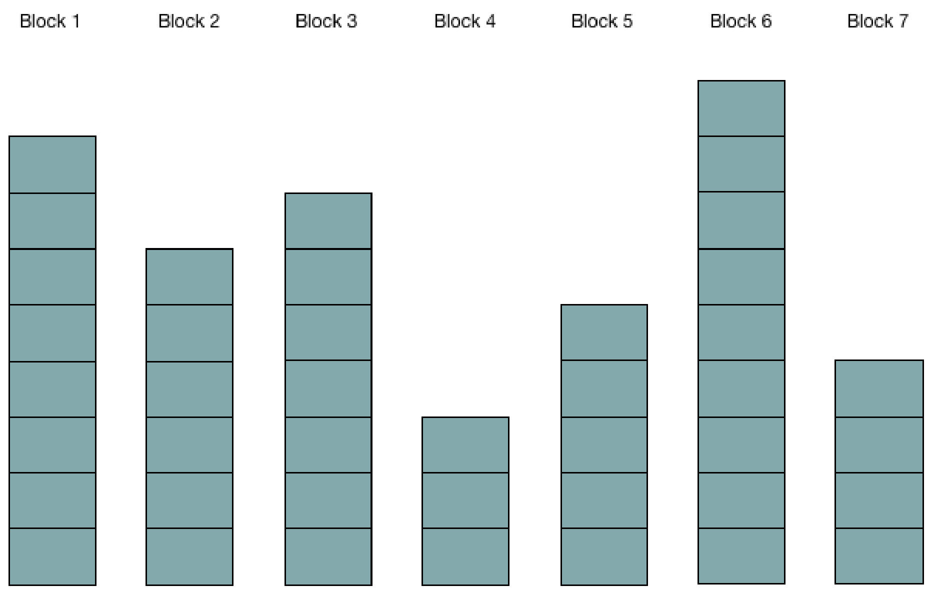

Figure 7: Hypothetical window of blocks.

The most natural approach for transmission would be to transmit Block 1 first. Then, send Block 2, and so on. However, this is not how the Bitcoin Satellite transmitter works. Instead, the transmitter continuously iterates over a window of blocks while transmitting a single chunk out of each. The process becomes as follows:

On the first iteration, the transmission loop in this example sends one chunk from each of the seven blocks. In the next iteration, it repeats the process. More generally, it continues transmitting one FEC chunk out each of each block in the window indefinitely. In other words, the blocks are sent in parallel instead of in series.

Eventually, when a block is fully transmitted, the node adds a new block to the Tx block window. Thus, the transmission window keeps moving. For example, in the illustrated above, the first block that would complete transmission would be *Block 4*, the smallest block in the window (which only has three chunks).

> Note that the FEC chunk size is fixed (of 1152 bytes), and it is the number of chunks that varies among blocks. While tiny blocks can be split into a few fragments, large ones can use more than a thousand chunks (sometimes up to two thousand).

The primary motivation for the interleaved transmission approach is to protect from error bursts and outages of the receiving end. The benefit of doing so is better explained with an example.

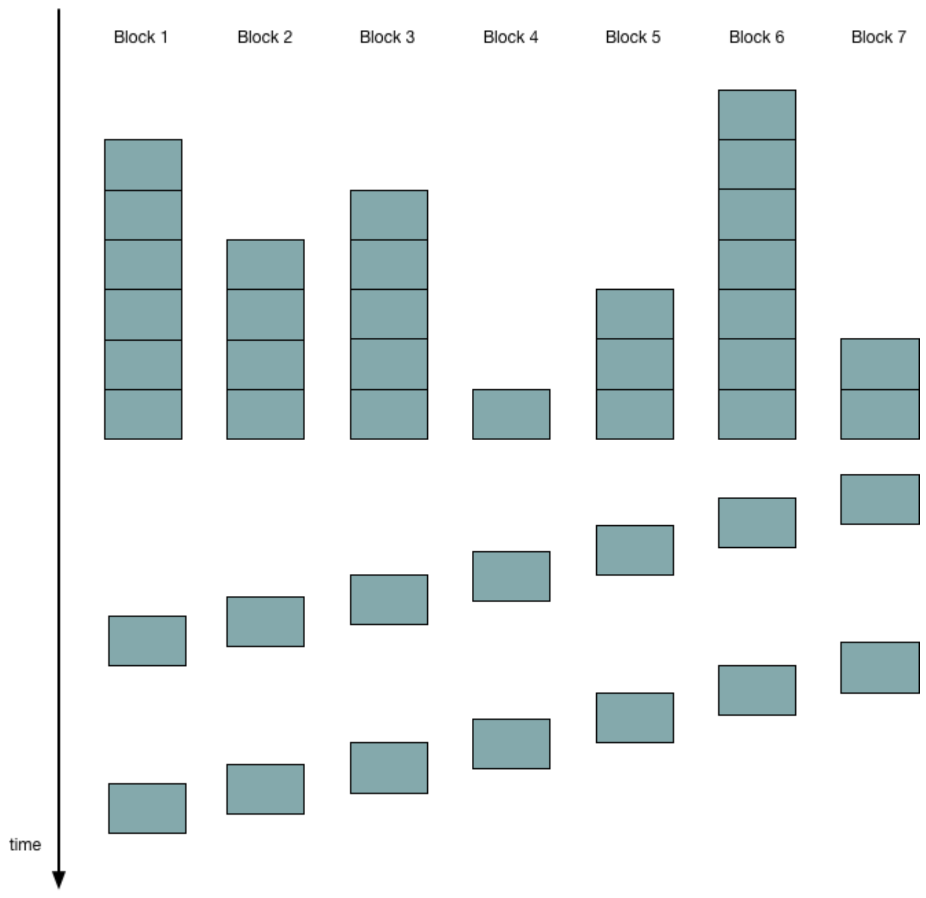

Figure 8: Time-interleaved transmission of FEC chunks from a window of blocks.

Suppose that the transmitter sends a single chunk per second. Next, suppose this transmitter is sending an FEC-encoded block composed of 250 FEC chunks, 50 of which are redundant (i.e., `y=250` and `x=200`). Finally, suppose that a receiver suddenly experiences a complete satellite outage for 5 minutes (i.e., 300 seconds). If the sender node did not interleave the chunks, the receiver would miss 300 chunks, namely the 250 pieces of the given block, including the excess 50 fragments. As a result, the receiver would not be able to recover the referred block.

Now, contrast this with the case where the transmitter alternates between chunks of 1000 consecutive blocks. The hypothetical transmitter that sends a single chunk per second would take 1000 seconds to transmit a single fragment out of each block in the 1000-block window. Consequently, if a receiver experiences a 5-minute outage, it loses a single chunk from 300 distinct blocks, rather than potentially 300 chunks of the same block. Finally, given that each block has excess chunks (the example block has 50), the receiver can still recover the data in this case. If the receiver misses one chunk of the hypothetical block with 50 excess chunks, there are still 49 extra chunks to back up the decoding process. In other words, the interleaved approach distributes the chunk losses among many blocks.

The only drawback of the interleaved transmission approach is that it leads to longer intervals to transport each block over the multicast-addressed UDP streams. However, in the long term, the average block transmission speed is equivalent. The interleaved approach takes longer to complete the transmission of individual blocks because many blocks are effectively transported in parallel.

Lastly, note that Stream 1 (which carries recently mined blocks) does not interleave chunk transmissions. The rationale is that Stream 1 focuses on transporting a single block (a new block in the chain) as quickly as possible. Thus, the interleaved transmission approach does not apply to it. Moreover, Stream 1 benefits from extra redundancy, as explained later, so it needs less protection than the blocks sent over Stream 2 and 3. Besides, if a receiver fails to receive the new block sent over Stream 1, it can still get the repetitions later over Streams 2 and 3.

1.4.4 Out-of-order Block Processing

Another essential element for achieving a reliable transport of blocks over Bitcoin Satellite is the processing of out-of-order blocks. All blocks in the blockchain have a number since the Genesis block. This number is the so-called block height. A block is said to be *in order* when its height is equal to the previous block height plus one. Otherwise, the block is out of order. For example, if the previous block has a height of `640299` and the node subsequently receives height `640302`, the latter is out of order.

As explained earlier, some streams send several blocks in parallel (with time-interleaved FEC chunks). Hence, on a window of blocks sent in parallel, the shorter blocks tend to complete earlier. This means that the blocks are not received in sequence. This parallel transmission feature is the first cause of blocks received out of order by a Bitcoin Satellite Rx node.

Next, recall that Stream 1 sends new blocks as they arise in the p2p network. Therefore, Stream 1 always transmits the blocks in sequence. The only exception is when there is a chain reorganization.

Nevertheless, an Rx node can fail to receive a block transmitted by Stream 1 and then wait for its repetition sent later through Streams 2 and 3. While the repetition does not arrive, the node will continue to receive new blocks via Stream 1. From this point on, all blocks received via Stream 1 become effectively out-of-order blocks because they are on top of a height that is still missing. This is the second reason why blocks can be received out of order.

Lastly, there is a third scenario where blocks become out of order. Recall that Bitcoin Satellite propagates the entire blockchain through Stream 3. Thus, an Rx node may start with an empty Bitcoin data directory and achieve full synchronization while disconnected from the internet. However, the problem is that when the Rx node is launched, the Blockstream Satellite Tx node may be transmitting any block, not necessarily the first block. Thus, the Rx node can start from any height within the chain. As a result, the node receives the entire chain out of order until it catches up with the chain's start.

Bitcoin Satellite overcomes the three referred scenarios by saving out-of-order blocks (OOOBs) on a dedicated database. Any block received out of order via satellite (more specifically, using the `-udpmulticast` option on `bitcoind`) is first minimally validated. If it passes this validation, it is then stored on the OOOB database. Correspondingly, whenever a block is processed in regular order, Bitcoin Satellite checks if the subsequent blocks (to be placed on top of the incoming block) are available on the OOOB database. In the positive case, the Bitcoin Satellite Rx node processes the succeeding blocks immediately.

1.5 Data Structures Used to Transport Blocks over Bitcoin Satellite

Before proceeding to the next topic, it is worth clarifying the data structures sent over multicast-addressed UDP streams. For each block, the Tx node sends two structures (interchangeably called *data objects*): the *header* and the *body* of a block. The Tx node transmits the header object first, then sends the body object. The two objects are FEC-encoded separately. Correspondingly, the Rx node decodes the two objects independently.

The header object contains the standard block header and auxiliary information. More specifically, it consists of a modified version of the `cmpctblock` message containing the `HeaderAndShortIDs` structure. The modified version is called `HeaderAndLengthShortTxIDs`.

Among other information, the `HeaderAndShortIDs` structure has a field called `shortids`, which contains the short transaction IDs of each transaction (txn) in the block. The modified message sent by the Bitcoin Satellite Tx node (again, called `HeaderAndLengthShortTxIDs`) has the same information and more. For instance, it includes a field named `txlens`, which consists of a list with the length of each txn in the block. Ultimately, upon reception of the *header* object, the Rx node becomes aware of the txns in the block that is yet to come.

The *body message* contains the actual txns, i.e., the `transactions` list from the regular block structure. The entire list is treated as a single data object and split into multiple chunks (up to thousands). Then, as mentioned earlier, the FEC encoder post-processes all the `x` original chunks and generates `y` encoded chunks with redundancy. The body object comprises these `y` chunks.

The processing sequence on the Rx node is as follows. For each block, the node starts by receiving the *header FEC chunks*. Once it has enough header chunks, it decodes the header object. Then, with the header information, the node can prepare the block that is yet to come. Subsequently, the node receives the *body FEC chunks*. Once it has enough body chunks, it decodes the body message. At this point, the node completes the block information and processes the block as usual through Bitcoin Core mechanisms.

1.6 How Bitcoin Satellite Improves Block Transmission Speeds

Bitcoin Satellite has a mechanism to transport new blocks recently mined in the p2p network as fast as possible. The scheme comes from Bitcoin FIBRE's implementation. It is based on the transmission of mempool transactions (txns) over the UDP multicast streams.

Each Bitcoin block contains a series of txns. Hence, when the receiver has some txns of the block ahead of time, it can fill portions of the block in advance. In the context of FEC-encoded transmissions, the receiver (possessing some txns of the block) can fill some FEC chunks of the block *body object* before actually receiving the chunks.

The Bitcoin Satellite Tx nodes continuously send the mempool txns that are more likely to enter new blocks (yet to be mined). Thus, satellite receivers worldwide receive and store txns even if they are connected only via satellite.

When a new block finally arises in the p2p network, the Tx node starts by transmitting the corresponding *header object* over Stream 1. Once an Rx node receives and decodes the header object (containing the `HeaderAndLengthShortTxIDs` structure), it can determine the block's exact memory layout. With that, the

Rx node immediately reserves memory for the block's *body object* and prefills all the txns that it already has in its local mempool.

As shown below, the prefilling operation implies that some of the body FEC chunks become known in advance. The remaining FEC chunks containing parts of txns that are not available locally yet become erasures. Consequently, when the Tx node starts to transmit the body object (sent after the header), the Rx node knows most of it already. In this case, the Rx node only waits for a few remaining FEC chunks.

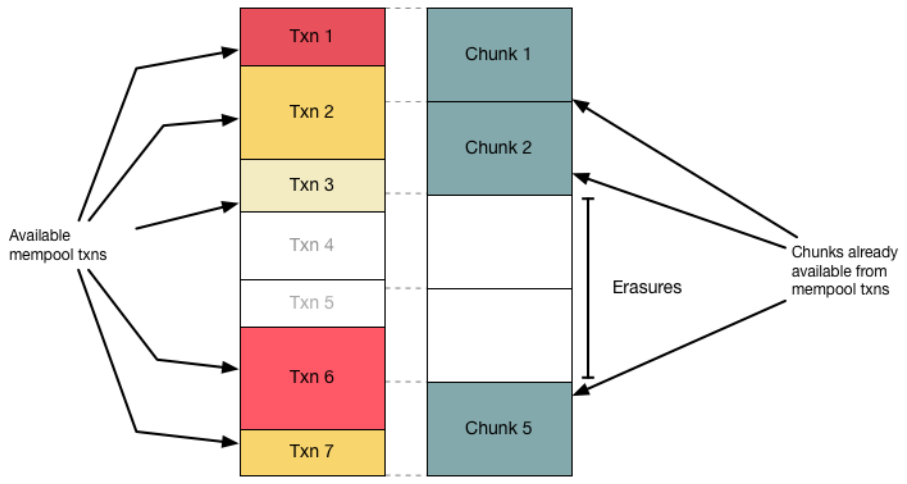

Figure 9: Pre-filling of FEC chunks based on txns available on the local mempool.

Finally, recall that an erasure coding FEC mechanism can recover the original data from any set of `x` unique FEC-coded chunks. Since the Tx node sends FEC chunks with random FEC IDs, it becomes very likely that every incoming chunk is a unique (new) chunk to the Rx node. In the end, if the receiver prefills `z` FEC chunks based on the local (mempool) txns, only `x-z` FEC chunks remain as erasures. Thus, the Rx node only needs to acquire `x-z` chunks to complete the FEC decoding. This mechanism provides a substantial reduction in the overall block transport latency.

Furthermore, this mechanism results in extra redundancy (protection) on the transport of new blocks over Stream 1. If the Rx end already has `z` chunks of a block and only needs `x - z` remaining chunks, while the Tx end sends `y` FEC chunks, it means that all `y - (x - z)` chunks become redundant. Recall that in Streams 2 to 4, each block has `y - x` redundant chunks. Here, in contrast, on Stream 1, each block has `y - x + z` excess chunks. For `z > 0`, this is a more significant number of extra chunks.

Suppose, for instance, the case of a block with `x=1000` chunks (again, sent with `y=1110` FEC chunks). If the Rx node can prefill `z=950` chunks out of `x=1000`, it only needs to receive `50` remaining chunks. In this case, the latency to download the block is 20 times lower than if it were to be received with no prefilling mechanism.

1.7 Compressed Transactions

Another essential feature of Bitcoin Satellite is that it uses a more efficient representation of txns with compression. The Bitcoin Satellite Tx nodes send FEC-encoded blocks with compressed txns. The Bitcoin Satellite Rx nodes, in turn, decompress the txns after decoding a FEC block. Most txns are compressed by around 20%, which results in significant savings on data transmissions.

1.8 Monitoring a Bitcoin Satellite Rx Node

So the next question is how one can verify that a Bitcoin Satellite Rx node is operating correctly. In general, you can check if the node is staying in sync with the blockchain and receiving multicast-addressed UDP data via the satellite receiver. You can check this using RPC commands and by inspection of logs collected by the `debug.log` file in your Bitcoin data directory (by default at `~/.bitcoin`). This section explains how.

The explanation that follows applies to Bitcoin Satellite version `v0.19.1.0.2.0` (read as satellite version `0.2.0` based on Bitcoin Core `0.19.1`) or any later version. You can check your version by running:

```
bitcoind --version
```

Version `v0.19.1.0.2.0` shows the following:

```
v0.19.1.0-g7fb5a08899ed6dc34229fb8476d2b1666f076fc8
```

If you would like to update Bitcoin Satellite to the latest version, please refer to blocksat-cli's documentation.

1.8.1 Monitoring the Satellite Traffic

Firstly, you can run the RPC command `getudpmulticastinfo` to obtain information from the UDP Multicast group configured for satellite reception. For instance, run:

```
bitcoin-cli getudpmulticastinfo
```

The result includes the traffic's measured bitrate.

```
{
  "172.16.235.9:0": {
    "bitrate": "1.060615 Mbps",
    "group": 0,
    "groupname": "blocksat-tbs",
    "ifname": "dvb0_0",
    "mcast_ip": "239.0.0.2",
    "port": 4434,
    "rcvd_bytes": 235730221924,
    "trusted": true
  }
}
```

1.8.2 Monitoring the FEC decoding process

You can also monitor the status of the FEC decoding process. Bitcoin Satellite has an RPC command called `getchunkstats`, which can be executed as follows:

```
bitcoin-cli getchunkstats
```

This command returns the number of blocks being download in parallel and the corresponding number of FEC chunks, as follows:

```
{
  "n_blks": 322,
  "n_chunks": 104637
}
```

Alternatively, when the block heights are already known, the command returns progress information concerning the blocks with minimum and maximum heights among the ones being received, as follows:

```
{
  "min_blk": {
    "height": 640064,
    "header_chunks": "22 / 22",
    "body_chunks": "251 / 930",
    "progress": "28.68%"
  },
  "max_blk": {
    "height": 640067,
    "header_chunks": "19 / 19",
    "body_chunks": "706 / 938",
    "progress": "75.76%"
  },
  "n_blks": 1650,
  "n_chunks": 623947
}
```

However, note that the Rx node first needs to decode the header object (with the `HeaderAndLengthShortTxIDs` structure) to become aware of the incoming heights, as explained earlier. Thus, the `getchunkstats` RPC command typically returns the former result, containing the number of blocks and chunks only.

Furthermore, as soon as the node decodes the header object, where it finds the block height information, it can check whether the block is needed or already available locally. If the block is already available locally, the node stops processing the block. Any further FEC chunk received for this block gets dropped to avoid unnecessary work.

Ultimately, when the node is in sync with the blockchain and operating for a sufficiently long interval, command `getchunkstats` commonly returns zero blocks and chunks, as follows:

```
{
  "n_blks": 0,
  "n_chunks": 0
}
```

This is a valid result when all blocks being received over satellite are already available locally. However, as soon as the node starts to receive a new block (one that is not available locally yet), the counts become non-zero.

An alternative view can be obtained by running `getchunkstats` with argument `0`, as follows:

```
bitcoin-cli getchunkstats 0
```

This command returns information from all the FEC-encoded blocks that are currently partially received.

```
{
  "0035f332c7544c4d": {
    "header_chunks": "9 / 14",
    "body_chunks": "599 / 835",
    "progress": "71.61%"
  },
  "00418a9a3d82cde8": {
    "header_chunks": "19 / 20",
    "body_chunks": "498 / 812",
    "progress": "62.14%"
  },
  "00773ee8aee3efa6": {
    "header_chunks": "0 / 0",
    "body_chunks": "268 / 483",
    "progress": "0%"
  },
  ...
```

The key of each entry corresponds to the 64-bit prefix of the block hash. Each entry shows the number of chunks received for the *header* and the *body* FEC objects. When the node has not received any header or body chunk, the returned count remains `0 / 0` because the total number of chunks composing the object is still unknown. The progress percentage is determined based on both header and body chunks.

1.8.3 Monitoring the Txn Hit Ratio

There is also an RPC command to get a sense of the block transfer speed gains due to the aforementioned txn prefilling mechanism.

The metric of interest is the so-called *txn hit ratio*, determined as follows:

```
fec_hit_ratio = txns_available_locally
                ----------------------
                     total_txns
```

That is, the txn hit ratio is given by the number of txns already available locally (in the mempool) when a block comes, divided by the total number of txns composing the block. A similar metric is the *FEC chunk hit ratio*, which is the ratio between the number of FEC chunks prefilled based on local txns and the total number of FEC chunks composing the block. The higher these ratios, the better it is for latency. A 100% FEC chunk hit ratio implies an immediate FEC decoding upon a block header message's reception.

You can monitor these ratios using the `getfechitratio` RPC command. For instance, run:

```
bitcoin-cli getfechitratio
```

The result shows both ratios separately, as follows:

```
{
  "172.16.235.9:0": {
    "txn_ratio": 1,
    "chunk_ratio": 1
  }
}
```

1.8.4 Observing Out-of-Order Block Receptions

As explained earlier, Bitcoin Satellite deals with blocks received out-of-order. You can check how many out-of-order blocks are waiting to be processed by running the following RPC command:

```
bitcoin-cli getoooblocks
```

Note that if you are running the full synchronization, there can be a large number of out-of-order blocks until your node catches up with the preceding blocks.