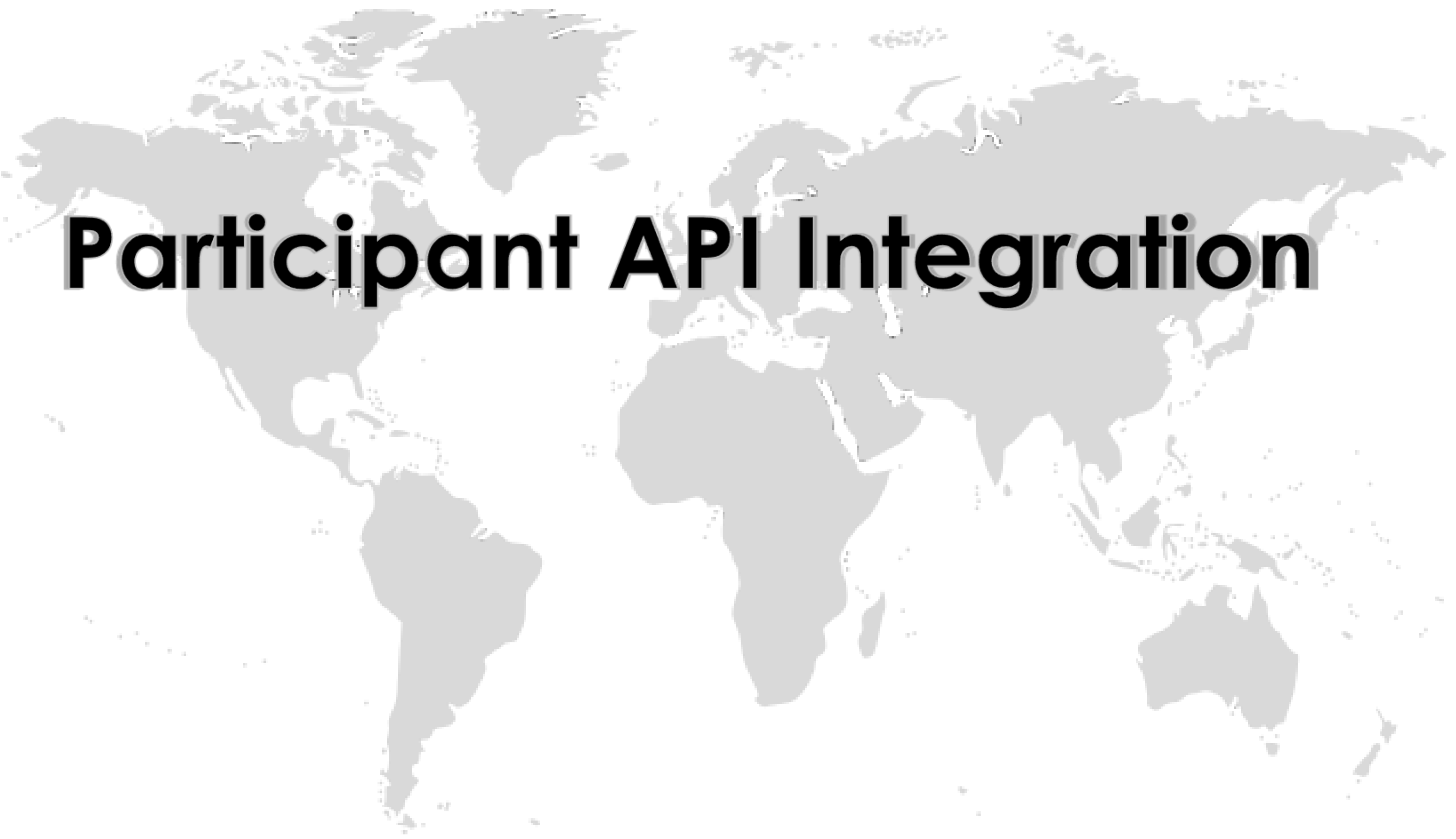
MONTRAN

National Bank of Georgia



Participant API Integration

Instant Payments System

Version: 1.00

Date: 2025-01-03

Document control

Title:	Participant API Integration
Code:	GE_IPS_Participant_API_Integration
Project:	GE_IPS_2023
Confidentiality:	BUSINESS USE ONLY
Integrity:	HIGH
Availability:	MEDIUM
Deliverable:	No
Version:	1.00

Document ownership

Nature of involvement	Name	Institution	Role
Owned by:	Sebastian Stefan	Montran	Technical Team Lead
First Draft by:	Sebastian Stefan	Montran	Technical Team Lead
Last Verified by:			
Reviewed/QA by:			
Approved by:			
Distributed to:		NBG	

Document history

Date	Version & Status	Changes
2024-12-30	0.01 DRAFT	First draft.
2025-01-03	1.00	Final version sent to NBG.

1. Participant API Integration

The Instant Payment System was designed to facilitate real-time transfers of funds between participants with a high availability (24/7/365). IPS provides immediate confirmation of payment, which is irrevocable and final, making the funds available to the recipient almost instantly (typically within seconds).

To match these characteristics, a synchronous communication protocol between IPS and the participants' systems has been defined based on HTTPS API. The participant system discussed here is the participant's own internal application, a payment message generation and processing solution which interfaces with the IPS API, called "IPS Connectivity Module" in the rest of the document.

To achieve real-time data transfers with high performance in the end-to-end payment flow, there must be a tight integration between the systems and an efficient implementation of the participants' IPS Connectivity Module. The API clients can be implemented in any programming language or alternatively, the dedicated Java IPS client library can be imported.

When implementing the IPS communication protocol the participants should consider the following recommendations for the IPS Connectivity Module:

1.1. Scalability

The IPS Connectivity Module can scale up by using multiple connections and processing threads in parallel and increasing the pool sizes as needed.

1.1.1. Connection pooling

Frequent HTTPS connections can introduce overheads due to the cost of repeatedly establishing connections. For this reason, the usage of an HTTPS connection pooling mechanism, which manages and reuses connections for multiple requests, is recommended. This approach offers better performance, because it reduces the frequency of time-consuming steps such as TLS handshakes and uses the available resources more efficiently.

There are multiple out-of-the-box implementations of connection pooling, therefore it is recommended to use the one already included in the chosen HTTPS client library (if available). The standard IPS client library uses the connection pooling mechanism from 'Apache HttpClient' library.

When choosing and configuring the connection pool, the following aspects should be considered: thread safety (the implementation must be thread-safe so it can be used in a multi-threaded context and allow for scalability), pool size (the pool size should be set based on expected traffic and available resources), idle timeouts (in case of variable traffic, timeouts can be set in order to avoid keeping unnecessary open connections and conserve resources).

Example for configuring connection pooling with Apache HttpClient (other configuration details omitted):

```
import org.apache.http.impl.conn.PoolingHttpClientConnectionManager;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
…
PoolingHttpClientConnectionManager connectionManager = new
PoolingHttpClientConnectionManager(…);
connectionManager.setMaxTotal(<poolSize>);
connectionManager.setDefaultMaxPerRoute(<poolSize>);

CloseableHttpClient httpClient =
HttpClients.custom().setConnectionManager(connectionManager)…build();
```

If using the Java IPS client library then only the following property must be set in the configuration file “client-config.properties”: MAX_CONNECTION_POOL=x, where x is the connection pool size.

1.1.2. Multithreaded processing

Besides connection pooling, the IPS Connectivity Module can scale by processing multiple messages in parallel. The HTTPS client should be thread safe, according to the previous section, which enables the client application to send and receive messages from multiple threads as well. Keeping a processing thread pool is necessary to parallelize time consuming operations such as XML parsing, digital signatures application and validation, and beneficiary account validation.

1.2. HTTPS Connection Management

In addition to maintaining a connection pool, the HTTPS client configuration can also be tweaked for optimal performance:

Socket timeout – the time limit for waiting to receive data after a connection has been established. It should be set low enough to detect contingency situations (to not keep the client stuck waiting for a response) and high enough not to interfere with normal transaction processing. For example, for a timeout of 10 seconds, all transactions that take longer to complete will be finalized with the asynchronous business flow (with an investigation business message or a timeout notification from IPS).

Connect Timeout – the timeout until a connection is established. It should be set to fail quickly and not keep the client hanging if the server is unreachable. This shouldn’t occur during normal processing, but the error message can help troubleshoot the issue.

Connection Request Timeout – the time limit for waiting when requesting a connection from the pool. In case if all the connections are used up and busy this timeout can be used to stop accumulating more requests that cannot be processed. It can be set similar to the SLA since it would be useless to send payments after SLA expiration.

Keep-alive time – when using connection pooling, the client must add “Connection: keep-alive” header to indicate to the server to not close the connections. The keep-alive duration should be enough so that connections are not closed too fast if there is a short idle interval. The default keep-alive duration in the IPS Java library is set to 60 seconds.

Example of setting the Keep-alive strategy for Apache HttpClient:

```
import org.apache.http.conn.ConnectionKeepAliveStrategy;
…
     static class KeepAliveStrategy implements ConnectionKeepAliveStrategy {

      @Override
      public long getKeepAliveDuration(HttpResponse response, HttpContext
context) {
             return 60 * 1000;
      }
}
...
httpClient = HttpClients.custom()….setKeepAliveStrategy(new
KeepAliveStrategy()).build()
```

Compression – the IPS Connectivity Module should be able to read and write compressed messages. IPS will compress messages (included in the request body) if the following header is present on client request: Accept-Encoding: gzip. Some HTTPS client implementations have this mechanism enabled by default (also enabled for the IPS Java library).

Example of adding the gzip header on requests made with Apache HttpClient:

```
import org.apache.http.Header;
import org.apache.http.HttpHeaders;
…
Header header = new BasicHeader(HttpHeaders.ACCEPT_ENCODING, "gzip");
List<Header> headers = new ArrayList<Header>();
headers.add(header);

CloseableHttpClient httpClient =
HttpClients.custom()….setDefaultHeaders(headers)...build();
```

1.3. Version Management

A rolling update mechanism is recommended to maintain the high availability guarantees for users. The downtime of one participant can impact all the other participants as well if there are transactions in which the offline participant is the creditor. Neither IPS nor the IPS Connectivity Module should have regular downtime for maintenance. Rolling updates allow administrators to update their systems with minimal downtime, by switching processing to other application instances while performing maintenance.

Briefly, in order to implement rolling updates, the application must be able to run in multiple instances simultaneously. During the planned update, a subset of instances is taken out of the active instances list, or stopped, marked offline. These instances are then updated, restarted and added back to the active list. If the updated instances are healthy, then this process is repeated with the other instances. If the update fails, then the application is still running on the instances with the older version while the new version is being fixed.

1.4. Efficiency

XML parsing

The recommended method for parsing XML is a SAX (Simple API for XML) library. Compared to the other types of XML parsers such as DOM (Document Object Model, which loads the entire documents in memory) or Object-based (which map XML directly to objects in the programming language, such as JAXB), SAX is faster and more efficient because it processes the document sequentially and discards unnecessary information.

Prioritization

SLA constrained messages should always be prioritized for processing. This ensures that large, informative messages such as reconciliations (camt.053) or exception flows such as recalls do not block the transaction flows and are processed whenever there are available resources. For example, if using a pool of threads for reading messages (from IPS or other systems) and a pool of threads for processing (producer/consumer pattern), then the messages can be passed between these pools using a priority queue. SLA constrained messages such as pacs.008 would be marked with a higher priority.

Blocking/Nonblocking IO

The IPS client library supports both blocking and nonblocking IO models for HTTPS requests. The underlying communication protocol is the same (the IPS API) but on the client side the thread sending the request can be blocked waiting for response or freed up to use CPU more efficiently. The blocking model is simpler and can be used to limit the number of initiated payments (if all processing threads are waiting for the final status, then no new payments can be prepared for sending). However, the nonblocking model is more efficient in terms of resources and can be used to increase performance.

1.5. Example of processing a spike of payments

The exact processing times and configuration details for each participant's IPS Connectivity Module can only be determined based on usage estimates and performance tests after IPS integration. The interactions between participants' internal systems (mobile applications, back office, IPS Connectivity Module) as well as their particular implementations are only presented to illustrate the example since the actual architecture and communication patterns can vary.

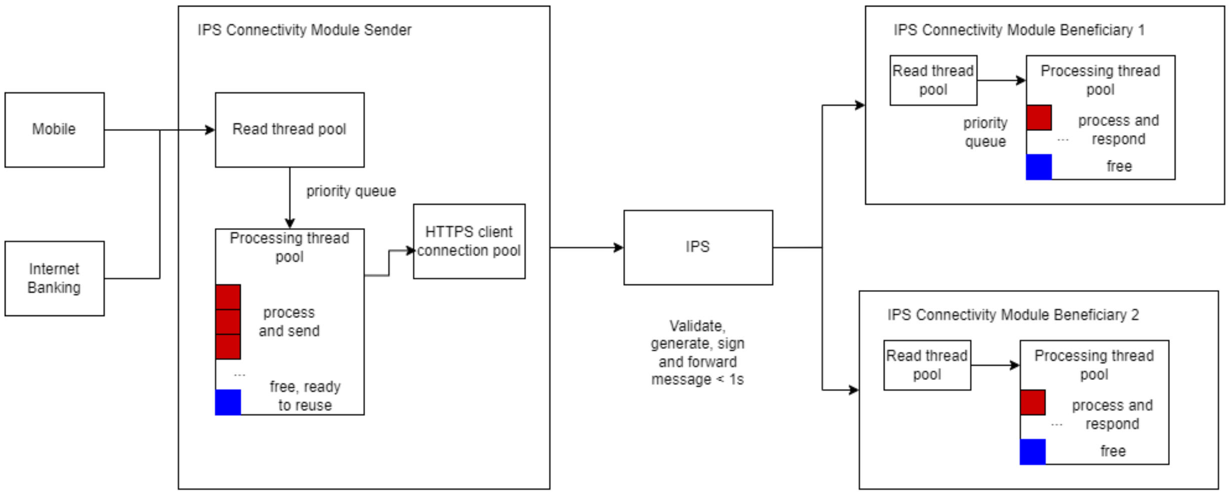
For example, with a connection pool size of 100, it means that there can be at most 100 in flight HTTPS requests at the same time. If we assume a round-trip time of about 1 second (the payment is sent to IPS, IPS validates and forwards it to beneficiary participant, beneficiary replies with a positive response, IPS forwards response to sender) this would be enough to achieve 100 TPS between the systems.

For the full flow involving the customer there are additional steps: the payment request is sent from the initial channel (internet banking, mobile application etc.) to the participant's IPS Connectivity Module, and the response from IPS is forwarded back to the initial channel and presented to the user.

Continuing with the example, if there is a moment when 200 users initiate transactions at the same time (outgoing payment flow):

- The requests are centralized in the participant's IPS Connectivity Module
- Assuming a processing thread pool of 20 workers, with enough cpu cores and an average of 20 ms required for internal message processing (XML message generation, store to db, etc.)
- After 20ms there would be 20 in flight requests, after 40ms there would be 40 requests etc.
- When the limit is reached (in this case 100) outgoing messages will be waiting for an available connection from the pool
- As soon as the HTTPS requests are completed the connections are reused, i.e. after 1 sec and 20 ms another batch of 20 requests will be sent to IPS

A diagram illustrating the message flows is presented below:



The IPS Connectivity Module presented here on both the sender and beneficiary sides can be the same application, since every participant is required to implement both ends of the flow. The internal pools and queues themselves can also be reused between the incoming/outgoing directions since they are both important.

In this example, the Mobile and Internet Banking Modules are the ones initiating a spike of transactions. The transactions are read into the IPS Connectivity Module then passed through a priority queue to the processing pool. The messages are forwarded to IPS through the HTTPS client connection pool. The same process is mirrored on the beneficiary side.

The completion of an end-to-end payment flow is therefore dependent on all these systems working together efficiently. Generally, most time in the payment flow is spent on the beneficiary side (XML parsing, beneficiary account validation, fraud check, anti-money laundering checks) and the participants must ensure these steps are done as efficiently as possible.

The initial values for the pool sizes, timeouts and other parameters presented in this document can be set based on the estimated volumes and the available hardware. Performance testing should also be done for additional tuning:

- if messages are being prepared much faster than they are forwarded to IPS than processing pool size can be reduced, and connection pool size can be increased
- if messages accumulate in the connection pool because of slow beneficiaries then the socket timeout can be reduced so that connection is not kept blocked (and the transaction are completed with the investigation flow later)
- if transactions timeout before they are consumed then the read pool can be increased
- if there is significant amount of time spend preparing the messages for IPS then nonblocking IO can be used to free up the processing pool while transactions are in-flight.