XCM: Cross-Consensus Messaging Audit

Technical audit report prepared for Parity

Reference	21-12-908-REP
Version	1.0
Date	2022/01/31

Quarkslab SAS
13 rue Saint Ambroise
75011 Paris
France

Contents

1 Project Information

Document history			
Version	Date	Details	Authors
1.0	2022/01/31	Initial Version	Robin David, Mahé Tardy

Quarkslab		
Contact	Role	Contact Address
Frédéric Raynal	CEO	`fraynal@quarkslab.com`
Stavia Salomon	Sales Manager	`ssalomon@quarkslab.com`
Robin David	R&D Engineer	`rdavid@quarkslab.com`
Mahé Tardy	R&D Engineer	`mtardy@quarkslab.com`

Parity		
Contact	Role	Contact Address
Benjamin Weiss	Director of Ecosystem Success	`benjamin.weiss@parity.io`
Shawn Tabrizi	Runtime Engineering Lead	`shawn@parity.io`
Keith Yeung	Core Runtime Developer	`keith.yeung@parity.io`

2 Executive Summary

This report describes the results of the security evaluation made by Quarkslab of the Cross-Consensus Messaging (XCM) mechanism developed by Parity for the Kusama/Polkadot blockchain. This message exchange format is the cornerstone of cross-chain communications between a relay chain and parachains, but also between parachains.

The main components to review are the XCM pallet and the XCM executor unleashing cross chain communications on a Polkadot network of chains. Parity provides multiple reference XCM pallet configurations for existing chains like Kusama Polkadot, Statemine, Westend etc. They are shared between the polkadot and cumulus projects. The audit focuses more specifically on the two base configurations (Kusama/parachain-template) and also more specifically on XCMv2 which is the latest version of the format.

The audit aims at checking the XCM implementation ensuring fairness between parachains and preventing any availability or consensus security issue. The main threats considered were: introducing inconsistencies between chain states, misconfigurations enabling unwanted asset transfers and asset discrepancies between chains.

This evaluation is the second audit performed on XCM, the first iteration having been performed by another audit company. The evaluation was carried by two auditors for a duration of 50 days. No major vulnerability has been found. Consequently, the report highlight the inner-working of XCM and security related intricacies making the code robust against aforementioned issues. Some design notes are described in the findings, see Section 2.2.

2.1 Disclaimer

This report reflects the work and results obtained within the duration of the audit on the specified scope (see. Section 3.3). Tests are not guaranteed to be exhaustive and the report does not ensure the code to be bug-free.

The security of XCM relies essentially on the used configuration parameters. As the audit focuses on the provided base configurations, any parachain changing these parameters exposes itself to security related issues. Regarding that, adaptating XCM behavior to the business logic of a parachain shall be done with extreme care.

2.2 Findings summary

Table 2.2 summarizes remarks and worth mentioning aspects of XCM behavior encountered during the course of the audit. They are detailed in the appropriate sections of the report. During the audit, some issues were in the process of being fixed, but no additional security-related issues have been found. Some warning blocks are also present in the sections of the report and highlight some potential issues or dangers.

ID	Description	Recommendation	Impact
INF01	timeout defined in QueryStatus::Pending is unused.	Check that timeout is lower than the current_block_number in on_response.	Info
INF02	On reception of answers for a pending query, check that the response type is what was expected is impossible.	Keep the expected response type in Queries and check that the type is correct on reception.	Info
INF03	Implicit link between expecting_response() and AllowKnownQueryResponses barrier.	Moving expecting_response() in the QueryResponse instruction handler of process_instruction.	Info

3 Context and Scope

3.1 Context

As part of the Polkadot network deployment and the parachain slot auction process, Parity Tech aims at auditing its XCM component. It enables message exchanges between any chains on the network. This components will enable fungible and non-fungible asset transfer between chains as well as any call to foreign extrinsics.

XCM aims at transparently ensuring the consistency and finality between chains (i.e., an asset cannot be rolled-back on a chain while being considered final on the other chain) while also ensuring fairness between chains. Core abstraction principles are detailed by the Web3 Foundation [1]. Short and mid-term prospective overview in Polkadot are detailed by Gavin Wood [2].

3.2 Safety and Security Properties

In addition to security properties provided by the blockchain, a cross-chain communication mechanism introduces cross-consensus issues. Hopefully parachains are not fully independent and uses the consensus mechanism of the underlying relay chain. The block time can vary so there are no 1:1 block correspondance between relay chain and parachains. Yet the parachain mechanism ensures a perfect lineage tracking between relay chain and parachain blocks. Derived from such communication mechanism, main security properties that must hold are the following:

- finality: asset transfered, or teleported from one chain to another, must ensure the correct lock/unlock or burn/mint on their respective chains;
- fairness: each parachain must have an equal chance to communicate with the relay chain or another parachain.

The considered security model takes into account a misbehaving parachain, or one actively behaving in an adversarial manner against the relay or another parachain. Also, all extrinsics triggered by users are considered untrustworthy.

XCM messages can be emitted in two manners. First, they can directly be submitted by users via the XCM pallet. Otherwise, they can be emitted by the chain itself by means of executing another extrinsic. These two vectors are considered in this study and further described in Section 4.1.

3.3 Scope

The audit involves different software components scattered over different projects. The main component is Substrate[1], the base framework for building a standalone Proof-Of-Stake (PoS) blockchain. It provides base pallets to perform the main actions of a blockchain. Atop, `Polkadot`[2] provides additional features to build a network of chains through the parachain mechanism. It

[1] https://github.com/paritytech/substrate
[2] https://github.com/paritytech/polkadot

provides the XCM pallet. Then the `Cumulus` project[3] uses both of these projects and implements core functionalities of a parachain pluggable on a relay chain. The XCM behavior is configured differently by a relay chain and a parachain. The audit scope focuses on the following components and use-cases:

- relay chain XCM configuration in the Polkadot project
- parachain XCM configuration in Cumulus
- whole XCM pallet in Polkadot project
- asset transfers, reserve transfers, teleport transfers
- DMP, UMP, HRMP queue handling on the relay chain
- version negotiation

Some use-cases of XCM are yet to be implemented or already considered legacy. As such the audit leaves out-of-scope the following components:

- scale encoding/decoding
- XCM v0 and v1. While remaining present (and some types used), future parachains are meant to strictly forbid legacy XCM versions. As such the focus has been given to v2.
- plurality, and governance delegated to a parachain
- usage of XCM for relay-to-relay communications
- benchmark/weight computation

3.4 Methodology

The XCM pallet audit methodology has roughly been divided into the following steps:

- code and implementation discovery
- network configuration with 1 parachain (see Section 3.5 below)
- static code review (XCM executor, runtime configuration in polkadot and cumulus etc.) This includes covering the following aspects:
  - logic bugs in implementation and race-conditions (e.g: between extrinsics)
  - "costless-storage": storage not properly weighted (e.g: no existential deposit)
  - "costless-computation": improperly weighted extrinsic costs
  - usage of unsafe code constructs (e.g: unsafe arithmetic, potential panics, unwraps)
- dynamic testing of RC-PC, PC-PC communications
- report writing

[3] https://github.com/paritytech/cumulus

3.5 Audit Settings

As the XCM codebase is undergoing significant changes thanks to a prior audit perfomed on it, versions used for the audit have been frozen in accordance with the Parity team. Especially, the version of `Polkadot` and `Cumulus` have been fixed. For both of these projects the used reference chain specification is respectively Kusama and Statemine. Exact versions and commit-id are shown respectively in Table 3.1 and Table 3.2. From the specificed commit hashes, the two binaries were compiled and small modifications were applied to the `Polkadot` binary to reduce the `EPOCH_DURATION_IN_SLOTS` constant from `1*HOURS` to `2*MINUTES` to make the registration of the parachain faster for the Kusama runtime.

Project	`Polkadot`
Repository	`https://github.com/paritytech/polkadot`
Commit hash	`7d8f00b90cd6d87780123b3e08ca120cfb0c6e50`
Commit date	2021/12/02
Chain-spec	Kusama
Runtime	v0.9.13 *(runtime 9130)*

Table 3.1: `Polkadot` version references

Name	`Cumulus`
Repository	`https://github.com/paritytech/cumulus`
Commit hash	`0be8e8fc214641e306e4f913dd64ff1913e46e95`
Commit date	2021/11/24
Chain-spec	Statemine
Runtime	v6.0.0 *(runtime 600)*

Table 3.2: `Cumulus` version references

3.6 Polkadot-launch configuration

To start a network, the CLI tool Polkadot-launch[4] was used with a configuration to start a `kusama-local` with two validators and `statemine-local` with two collators network.

Small modifications were made to the default `statemine-local` chain specification, generated with the `build-spec` command of the collator binary, to increase the base amount of tests accounts of the `balances` pallet.

[4]`https://github.com/paritytech/polkadot-launch`

4 XCM pallet

4.1 Pallet XCM

`pallet-xcm` provides ten extrinsics that can be split into three categories. The first category provides primitive functions to execute locally or send an XCM message. The second provides high-level functions for asset transfers. The last one provide some extrinsics aimed exclusively at version negotiation.

Primitives

`execute`

This call is a direct access to the XCM executor. It checks the origin (see. 4.2), the message, and ensure no filter is blocking the execution. Then it executes the message locally and return the outcome as an event. It is necessarily executed on behalf of the account that have signed the extrinsic (origin).

The relay chain uses the `execute_xcm_in_credit` function of the XCM executor code base to execute locally. This function mainly tries to weight the message to see if the weight limit can be respected, checks if it passes the barrier(s), executes the message respecting the error handling and/or the possible appendix programs then finally handles surplus weight and returns the outcome of the execution to the caller.

`send`

The `send` extrinsic is an interface to send a message to a destination. By following the source code on `Self::send_xcm`, which trait implementation for a relay chain can be found in */runtime/common/src/xcm_sender.rs*. Listing 1 shows that for now, the only valid destination is exactly one parachain. Also, in this context "sending" means putting the message in a queue for downward destinations. These destinations will retrieve available messages in their queues and then execute it with their executor implementation.

```
match dest {
    MultiLocation { parents: 0, interior: X1(Parachain(id)) } => {
        // Downward message passing.
        let versioned_xcm =
            W::wrap_version(&dest, msg).map_err(|()|
↪   SendError::DestinationUnsupported)?;
        let config = <configuration::Pallet<T>>::config();
        <dmp::Pallet<T>>::queue_downward_message(
            &config,
            id.into(),
            versioned_xcm.encode(),
        )
        .map_err(Into::<SendError>::into)?;
        Ok(())
    },
    dest => Err(SendError::CannotReachDestination(dest, msg)),
}
```

Listing 1: `send_xcm` trait implementation

Transfers of assets

It is not possible to initiate transfers from a system to another without having the authority (and thus the origin) of the trusted system directly. That's why it is not possible to send a transfer message with a user account as origin, using the `send` extrinsic, because a `DescendOrigin` will be systematically inserted before any instruction added in a XCM message to reflect its real origin. For more information on this aspect see Section 6.4.

To make transfers between users across systems possible, the `pallet-xcm` proposes four extrinsics, or more precisely two extrinsics that will send messages with the origin of the sender system for some controlled parts of the message with some additional checks.

Indeed, there are really only two functions underneath, `do_reserve_transfer_assets` and `do_teleport_assets`. These functions are wrapped by four extrinsics:

- `teleport_assets`;
- `reserve_transfer_assets`;
- `limited_teleport_assets`;
- `limited_reserve_transfer_assets`.

The *limited* version of these wrappers provide the possibility to input another argument, an `Option<WeightLimit>` that will be passed to the actual functions. These weight limit, if not explicitely set, will be computed.

`teleport_assets` and `limited_teleport_assets`

The weight computation process can help to understand these extrinsics. First, the local weight is calculated based on the following XCM message:

```
Xcm(vec![
    WithdrawAsset(assets),
    InitiateTeleport { assets: Wild(All), dest, xcm: Xcm(vec![]) },
]);
```

Indeed, these four extrinsics execute some XCM messages and this is the part that will be executed locally. In that message, the nested XCM part in the `InitialeTeleport` instruction is empty.

Next, in the corresponding `do_teleport_assets` function, if no weight limit has been passed, the remote weight will be calculated by weighting the following message:

```
Xcm(vec![
    ReceiveTeleportedAsset(assets.clone()),
    ClearOrigin,
    BuyExecution { fees, weight_limit: Limited(0) },
    DepositAsset { assets: Wild(All), max_assets, beneficiary },
]);
```

Logically, the final message should be a combination of the two that were used to give an estimation of the local and remote weight.

```
Xcm(vec![
    WithdrawAsset(assets),
    InitiateTeleport {
        assets: Wild(All),
        dest,
        Xcm(vec![
            BuyExecution { fees, weight_limit },
            DepositAsset { assets: Wild(All), max_assets, beneficiary },
        ]);
    }
]);
```

And that's almost it, only two instructions `ReceiveTeleportedAsset` and `ClearOrigin` are missing from the nested message. It is because these two instructions will be prepended at execution time by the XCM executor. The code of the processor of the `InitiateTeleport` instruction specifies these two lines:

```
let mut message = vec![ReceiveTeleportedAsset(assets), ClearOrigin];
message.extend(xcm.0.into_iter());
```

To conclude, some instructions trigger the sending of another XCM message, and often, it needs to be interpreted with the sender system as origin instead of the user that initiated the message. That's why some instructions, like `ReceiveTeleportedAsset`, are injected and require some specific origin. Then a `ClearOrigin` or `DescendOrigin` will be added to prevent the user from being able to execute some controlled instructions with a superior level of privileges.

`reserve_transfer_assets` and `limited_reserve_transfer_assets`

These extrinsics are highly similar to their teleport counterparts. In this situation, the message that will be finally executed is the following:

```
Xcm(vec![
    TransferReserveAsset {
        assets,
        dest,
        Xcm(vec![
            BuyExecution { fees, weight_limit },
            DepositAsset { assets: Wild(All), max_assets, beneficiary },
        ])
    }
]);
```

And similarly, at execution, the `TransferReserveAsset` instruction will prepend `ReserveAsset-Deposited` and `ClearOrigin` before executing any nested instructions.

The implementation is secure as instructions like `TransferReserverAsset` require a privileged origin and the only way to obtain it is through these extrinsics that carefully downgrade the origin.

Version negotiation

The third category groups four extrinsics that deal with XCM version usage. They require `root` as origin so we can consider them as admin features. On Kusama, without the `sudo` pallet, it is impossible to call these extrinsics from “outside” the runtime.

`force_xcm_version`

This extrinsic is a root access to modify the `SupportedVersion` storage. It is a double map, containing the information about the version supported by destinations.

`force_default_xcm_version`

This extrinsic is a root access to set the `SafeXcmVersion` storage. This is the default version used when the destination version is unknown.

`force_subscribe_version_notify`

It sends an XCM message containing only a `SubscribeVersion` instruction to a destination. On a relay chain, it uses the `XcmRouter` directly so the final message contains only that instruction with the origin of the relay chain itself. It can also only be called as root.

`force_subscribe_version_notify`

This extrinsic is also sending an XCM message containing only one instruction, but this time `UnsubcribeVersion`. It can also only be called as root.

4.2 Pallet Configuration

The XCM pallet behavior can be configured by any runtime by modifying the `Config` of the pallet. It defines multiple traits that can be implemented. These parameters are very important. A bad configuration exposed a parachain to a hack in the past [3]. Listing 2 shows the Kusama configuration. Each type is discussed in the following sections.

```
impl pallet_xcm::Config for Runtime {
    type Event = Event;
    type SendXcmOrigin = xcm_builder::EnsureXcmOrigin<Origin,
↪   LocalOriginToLocation>;
    type XcmRouter = XcmRouter;
    // Anyone can execute XCM messages locally...
    type ExecuteXcmOrigin = xcm_builder::EnsureXcmOrigin<Origin,
↪   LocalOriginToLocation>;
    // ...but they must match our filter, which rejects all.
    type XcmExecuteFilter = Nothing;
    type XcmExecutor = XcmExecutor<XcmConfig>;
    type XcmTeleportFilter = Everything;
    type XcmReserveTransferFilter = Everything;
    type Weigher = FixedWeightBounds<BaseXcmWeight, Call, MaxInstructions>;
    type LocationInverter = LocationInverter<Ancestry>;
    type Origin = Origin;
    type Call = Call;
    const VERSION_DISCOVERY_QUEUE_SIZE: u32 = 100;
    type AdvertisedXcmVersion = pallet_xcm::CurrentXcmVersion;
}
```

Listing 2: Kusama XCM pallet configuration

Event, Origin and Call

These three types come from the `construct_runtime!` macro.

SendXcmOrigin and ExecuteXcmOrigin

`ExecuteXcmOrigin` is called in `execute`, `do_reserve_transfer_assets` and `do_teleport_-assets`. The `SendXcmOrigin` is only called in the `send` extrinsic. These functions are a more general way to authorize specific origins and convert them to `MultiLocations`. It is usually done per extrinsic using the `ensure_someOrigin` syntax, but this way makes a custom configuration easier for the entry points of sending and executing XCM.

```
/// Type to convert an `Origin` type value into a `MultiLocation` value which
↪  represents an interior location
/// of this chain.
pub type LocalOriginToLocation = (
    // We allow an origin from the Collective pallet to be used in XCM as a
↪   corresponding Plurality of the
    // `Unit` body.
    BackingToPlurality<
        Origin,
        pallet_collective::Origin<Runtime, CouncilCollective>,
        CouncilBodyId,
    >,
    // And a usual Signed origin to be used in XCM as a corresponding AccountId32
    SignedToAccountId32<Origin, AccountId, KusamaNetwork>,
);
```

Listing 3: Kusama origin to location conversion

`EnsureXcmOrigin` takes a converter as generic argument and Kusama uses one called `LocalOriginToLocation` shown in Listing 3. It is a tuple particularly composed of `SignedToAccountId32` that authorizes a signed origin to call the `execute`, `send`, teleport and reserve extrinsics.

XcmRouter

The XCM pallet needs an XCM router to be able to send messages. The Kusama relay chain uses a router called `xcm_sender::ChildParachainRoute`, ie. it is only able to send messages to child parachains at the moment. The Statemine Parachain for example uses a combination of two routers via a tuple, one to send to its parent relay chain, and one to send to siblings parachains.

Note

In the current state of development the router does not play a central role in terms of security but XCM is meant to play a broader role to bind different relay chains or any consensus system [2]. As such, it might play a greater security role when the topology of consensus systems connected via XCM becomes more complex.

XcmExecutor

It is the main component that contains the Cross-Consensus Virtual Machine (XCVM) and that will be used by the pallet to interprete instructions. It is further described in Section 5.

XcmExecuteFilter, XcmTeleportFilter and XcmReserveTransferFilter

These filters are called by their respective extrinsics, and it's the first filtering mechanism. They take a type that implements the trait `Contains`. `Everything` returns true and `Nothing` false, their deprecated counterparts are called respectively `AllowAll` and `DenyAll`, which can be more eloquent for filters. The Kusama configuration is (as the commentary implies) denying all attempts to execute a message. The `execute` extrinsic is somewhat disabled while transfers are allowed.

Weigher

Weigher are necessary to dynamically weight XCM messages that can be composed of many instructions and nested XCM messages. It is an important part to correctly adjust the fee that needs to be paid for the execution.

LocationInverter

The location inverter is used when sending XCM messages to reverse the references to assets. For instance, as all locations are relative in XCM, they must be modified when changing context.

Version Discovery Queue Size

The `VersionDiscoveryQueue` is an important part of the version negotiation further investigated in Section 6.2. This queue is popped by the `on_initialize` hook, called at each block, one element at a time. It is implemented as a bounded vector on Kusama with a limit of 100 elements. It means that this queue can contain a maximum of 100 destinations to request. This limit is not a concern because, first, its value is enough considering the number of different destinations in Polkadot or Kusama network. Secondly, because it would not imply any security concern if that queue was full, but only delay the version discovery process for some destinations.

AdvertisedXcmVersion

`AdvertisedXcmVersion` is the XCM version that will be advertised by the pallet when being asked, after receiving a `SubscribeVersion` by a peer wanting to communicate via XCM. In the current Kusama configuration, version 2 is used.

Conclusion

From a security perspective, many of these configuration settings are important, filters like `XcmExecuteFilter` are an easy way to enable or not parts of the pallet. The `ExecuteXcmOrigin` and `SendXcmOrigin` permit a more granular approach on the filtering of origins. Then the `Weigher` is crucial to correctly apply fees to the execution. Finally the `AdvertisedXcmVersion` is a constant and cannot be altered without modifying the runtime itself, which is good to avoid downgrades. The current configuration is restrictive which reduces the attack surface.

5 XCM Runtime Configuration

The XCM executor is parameterized by a `Config` trait defined in *polkadot/xcm/xcm-executor/src/config.rs*. It specifies sub-traits and functions for various components of XCM which behavior can be defined by the chain developer. Most functions of these traits are called within the executor upon receiving a specific instruction opcode. Hence, the configuration offers a gentle mechanism to program specific behavior for XCM instructions by implementing traits.

The overall security of a chain XCM executor relies almost entirely on the implementation of those traits[1]. In this section we focus on the Kusama configuration in *polkadot/runtime/kusama/src/lib.rs* and the base parachain-template configuration in *cumulus/parachain-template/runtime/src/lib.rs* for respectively the relay chain and the default parachain.

The following sections quickly describe the various traits, their usage in the executor and potential security implications.

5.1 AssetTransactor

```
/// How to withdraw and deposit an asset.
pub trait TransactAsset {
    // triggered by: ReceiveTeleportedAsset(MultiAssets)
    fn can_check_in(_origin: &MultiLocation, _what: &MultiAsset) -> XcmResult;

    // triggered by: ReceiveTeleportedAsset(MultiAssets)
    fn check_in(_origin: &MultiLocation, _what: &MultiAsset);

    // triggered by: InitiateTeleport(assets, dest, ...)
    fn check_out(_dest: &MultiLocation, _what: &MultiAsset); // InitiateTeleport

    // triggered by: DepositAsset(assets,..., beneficiary),
↪   DepositReserveAsset(assets, .., dest, ..)
    fn deposit_asset(_what: &MultiAsset, _who: &MultiLocation) -> XcmResult;

    // triggered by: WithdrawAsset(MultiAssets)
    fn withdraw_asset(_what: &MultiAsset, _who: &MultiLocation) -> Result<Assets,
↪   XcmError>;

    // triggered by: beam_asset()
    fn transfer_asset(_asset: &MultiAsset, _from: &MultiLocation, _to:
↪   &MultiLocation) -> Result<Assets, XcmError>;

    // triggered by: TransferAsset(assets, beneficiary)
    fn beam_asset(asset: &MultiAsset, from: &MultiLocation, to: &MultiLocation) ->
↪   Result<Assets, XcmError>;
}
```

[1]which vary from a chain to another

The trait defines functions configuring the withdrawal and deposit of assets. The implementation is parameterized by a currency, a means of converting a `MultiLocation` into an account, etc. The main difference between Kusama and Statemine is the `AccountIdConverter` which allows resolving a `MultiLocation` to an account.

Kusama configuration. It is configured to address both local accounts or child parachain accounts. It also defines a `CheckAccount` to hold native teleported assets which are not yet back on the chain.

parachain-template configuration. The `AccountIdConverter` allows converting relay chain, parachain sibling or local `MultiLocation` to a local account which is slightly different from Kusama. Nevertheless, it does not define a `CheckAccount` and thus does not track teleported assets.

Both of these configurations seem to flawlessly fulfill their usage.

5.2 OriginConverter

This trait defines a single function to implement for converting the origin on the behalf of which, the extrinsic call of a `Transact` instruction will be performed. Namely it converts a `MultiLocation` to an `OriginTrait` type.

Kusama configuration. It defines some acceptation criteria defined by:

```
type LocalOriginConverter = (
    // A `Signed` origin of the sovereign account that the original location controls.
    SovereignSignedViaLocation<SovereignAccountOf, Origin>,
    // A child parachain, natively expressed, has the `Parachain` origin.
    ChildParachainAsNative<parachains_origin::Origin, Origin>,
    // The AccountId32 location type can be expressed natively as a `Signed` origin.
    SignedAccountId32AsNative<KusamaNetwork, Origin>,
    // A system child parachain, expressed as a Superuser, converts to the `Root` origin.
    ChildSystemParachainAsSuperuser<ParaId, Origin>,
);
```

More precisely, given a `Transact` providing an `OriginKind`, the XCM executor takes the current origin and performs the following checks:

- `SovereignSignedViaLocation`: if the origin kind is `SovereignAccount`, it makes sure it can convert the current origin to a local account (`Origin::signed(id)`).
- `ChildParachainAsNative`: if the origin kind is `Native` make sure the current origin multi-location is a parachain. If so returns a `ParachainOrigin(id)`.
- `SignedAccountId32AsNative`: if the origin kind is `Native`, ensure the origin refers to a local account with the same network id. (`Origin::signed(id)`)
- `ChildSystemParachainAsSuperuser`: if the origin kind is `Superuser` and the current origin is a parachain which is **system** then enables it to perform the call as root (`Origin::root()`).

Parachain-template configuration. It defines the following criteria:

- `SovereignSignedViaLocation`: which attempts to convert the current origin to a local account whether it comes from the relay chain, a sibling parachain or a straight local account.
- `RelayChainAsNative`: if kind is `Native` and origin is the relay chain, perform the transact as the native chain.
- `SiblingParachainNative`: if kind is `Native` and origin is a parachain, convert to a `ParachainOrigin` origin.
- `SignedAccountId32Native`: same as Kusama
- `XcmPassthrough`: if kind is `Xcm`, then reuse the current origin

All these conversion criteria are important from a security perspective as they translate an XCM origin to a local origin specific to the current runtime. It somehow works like the discretionary access control (DAC) of Linux systems, where parachain developers are in charge of allowing or not some origins to execute a given extrinsic or code portion.

In a security model where a parachain fully trust the relay chain, the current configuration is adequately configured.

5.3 IsReserve and IsTeleporter

These configuration elements must implement the `FilterAssetLocation` trait that defines the following function:

```
fn filter_asset_location(asset: &MultiAsset, origin: &MultiLocation) -> bool;
```

It must return whether it accepts the `origin` as a reliable source for respectively a reserve or teleported assets. The reserve check is performed when receiving a `ReceiveAssetDeposited` instruction and the teleport when receiving a `ReceiveTeleportedAsset`.

Kusama does not accept a reserve but accepts teleports of any fungible token from the Statemine parachain. Conversely, parachain-template disables teleporting but defines `NativeAsset` as a reserve. That filter ensures the asset is managed by the origin of the message. Under this setting the parachain can only accept assets from the relay chain and the relay chain teleport assets from Statemine which is considered trusted.

5.4 LocationInverter

The trait defines the mean of inverting a `MultiLocation`. The framework provides a single implementation parameterized by the `Ancestry` which is `Here` for a relay chain and is `Parachain` for a parachain.

5.5 Barrier

Barriers are a set of filters to statically accept or reject XCM messages. It strongly influences the kind of messages that can be received by a chain. A barrier trait should implement the following function:

```
pub trait ShouldExecute {
    fn should_execute<Call>(origin: &MultiLocation, message: &mut Xcm<Call>,
↪   max_weight: Weight, weight_credit: &mut Weight) -> Result<(), ()>;
}
```

The function takes the origin that initiated the message, the message and weight-related parameters. Kusama is instanciated with the following barriers:

```
pub type Barrier = (
    TakeWeightCredit, // Weight that is paid for may be consumed.
    AllowTopLevelPaidExecutionFrom<Everything>, // If the message is one that
↪   immediately attemps to pay for execution, then allow it.
    AllowUnpaidExecutionFrom<IsChildSystemParachain<ParaId>>, // Messages coming
↪   from system parachains need not pay for execution.
    AllowKnownQueryResponses<XcmPallet>, // Expected responses are OK.
    AllowSubscriptionsFrom<OnlyParachains>, // Subscriptions for version tracking
↪   are OK.
);
```

More precisely they perform the following actions:

- `TakeWeightCredit`: Does not perform any checks on the message but instead checks that there are more credit allocated than the weight of the message. In the current implementation it only allows local execution of teleports and reserve transfers using `(limited_)teleport_asset` and `(limited_)reserve_transfer_asset` extrinsics. Indeed these functions call `execute_xcm_in_credit` with a specific credit equal to the message weight. All other means of getting in `execute_xcm_in_credit` is done without credit and will not be accepted by the barrier.
- `AllowTopLevelPaidExecutionFrom`: configured with `Everything`, it enables any origin as long as the message is of the form:
  - One of `ReceiveTeleportedAsset`, `WithdrawAsset`, `ReserveAssetDeposited`, `ClaimAsset`, followed by;
  - Zero or more `ClearOrigin` followed by;
  - `BuyExecution`.
- `AllowUnpaidExecutionFrom`: configured with `IsChildSystemParachain` grants any **system** parachain XCM message. Note that it does not perform any checks related to weight.
- `AllowKnownQueryResponses`: allows any message containing only a `QueryResponse` opcode and with an expected query ID.

- AllowSubscriptionsFrom: configured with OnlyParachains it only grants SubscribeVersion and UnsubscribeVersion as single instruction messages coming from parachains, allowing free version negotiation.

To recap, the barriers are part of the execution of XCM messages, they are early in the execution and answer the question “should this message be executed?”. They can be combined to allow exceptions, such as authorizing any messages that can be recognized as taking part to system functionalities, such as the version negotiation process with the subscription and query response messages, or the message coming from a trusted origin that should be allowed to execute anything. For example, the default parachain-template configures AllowUnpaidExecutionFrom with ParentOrParentsExecutivePlurality, thus allowing message from the relay chain or the executive plurality of the relay chain. Finally, its possible to implement more sophisticated barriers to check that a payment has been programmed into the message for the execution of standard user messages. Or, it can be used in more elaborate scenarios, such as ones with credits or a free tier of execution.

Barriers have the responsibility to accept or reject a message according to the chain policy. Their combinations should be made carefully in order for filters to reflect the chain business logic. Badly configured barriers can let unexpected messages to be blindly executed by the executor.

5.6 Weigher

A weigher is the mean of computing the weight for a given message. The builder *polkadot/xcm/xcm-builder/src/weight.rs* provides an implementation called FixedWeightBounds parameterized by a base weight for each XCM instruction opcode (10^9) and a maximum instruction number (100). Both Kusama and parachain-template are parameterized with the same value. A noticeable aspect of the algorithm is the weight of a call in a Transact instruction.

5.7 Trader

The Trader is the way to purchase the weight necessary to execute the message, it could in theory accept different assets and handle the conversion, but for now, the main implementation only handles the AssetId passed as a generic type, which is for Kusama and Statemine, the native coin of the relay chain. As explained in Section 5.5, the necessity to buy weight is enforced by the barriers, thus some messages can be executed without using the trader. The trait that a Trader must implement is the WeightTrader, which includes buy_weight and refund_weight methods shown in Listing 4.

```
/// Charge for weight in order to execute XCM.
/// [...]
pub trait WeightTrader: Sized {
    /// Create a new trader instance.
    fn new() -> Self;

    /// Purchase execution weight credit in return for up to a given `fee`. If
↪   less of the fee is required
    /// then the surplus is returned. If the `fee` cannot be used to pay for the
↪   `weight`, then an error is returned.
    fn buy_weight(&mut self, weight: Weight, payment: Assets) -> Result<Assets,
↪   XcmError>;

    /// Attempt a refund of `weight` into some asset. The caller does not
↪   guarantee that the weight was purchased using `buy_weight`.
    /// Default implementation refunds nothing.
    fn refund_weight(&mut self, _weight: Weight) -> Option<MultiAsset> {
        None
    }
}
```

Listing 4: `WeightTrader` trait

Functions `buy_weight` and `refund_weight` of the trader are only called in the executor by instructions `BuyExecution` and `RefundSurplus`. Only `refund_weight` is called via `refund_-surplus` at the end of the execution in the `ExecuteXcm` implemented in the `xcm-executor` library.

In Kusama, Statemine and the parachain template, the `UsingComponent` implementation is used, it can be found in *polkadot/xcm/xcm-builder/src/weight.rs*. It is a tuple struct that has three fields with types `Weight` and `Currency::Balance` associated with the trader. This implementation stores the fees paid in its second field and actually pays for the execution in its custom destructor code. Indeed `UsingComponent` implements the `Drop` trait that is explicitely called at the end of the execution of XCM messages.

In the current implementation, there are no means of getting more assets refunded, or tricking the trader in an inconsistent state.

Note

The `Trader` directly takes part in the fees payment process. For example, to pay for its own execution, a message typically includes:

- a `WithdrawAsset` instruction that will take assets from the author account and put it in the Holding registry;
- a `BuyExecution` instruction that will take a part of the assets in the Holding and give it to the trader.

Finally, at the end of the execution, some unused weight can be refunded and `drop` from the Drop trait will be called on the trader to deconstruct it and pay the system with the balance of the trader.

5.8 ResponseHandler

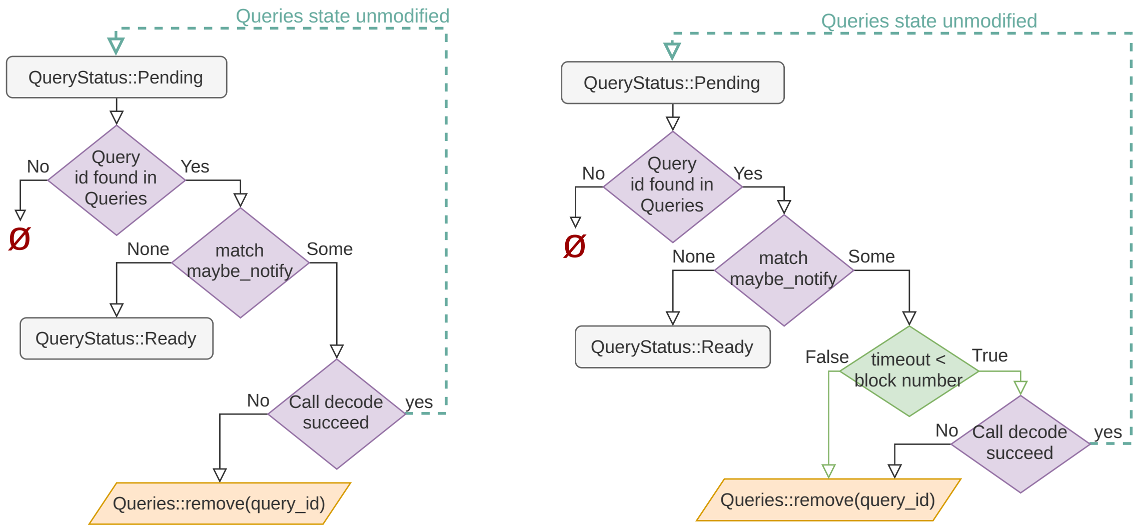
The `OnResponse` traits enables implementing defined actions when receiving `QueryResponse` instructions. Both Kusama and parachain-template use the same implementation provided by the XCM pallet. It checks that the response id is expected, and processes it when receiving it. It properly rejects any response that it did not query beforehand. Note that, for a certain type of pending query, if `maybe_notify` is not present, the query is never removed from the `Queries` storage which enables the origin of the response to send it indefinitely as barriers does not enforce buying execution for it.

Queries can have different `QueryStatus` associated in the `Queries` storage, see Listing 5. `QueryStatus::VersionNotifier` is dedicated to version negotiation subscription, `QueryStatus::Ready` is for queries already answered without any callback[2] specified, and finally, `QueryStatus::Pending` is for queries without response yet.

```rust
pub enum QueryStatus<BlockNumber> {
    /// The query was sent but no response has yet been received.
    Pending {
        responder: VersionedMultiLocation,
        maybe_notify: Option<(u8, u8)>,
        timeout: BlockNumber,
    },
    /// The query is for an ongoing version notification subscription.
    VersionNotifier { origin: VersionedMultiLocation, is_active: bool },
    /// A response has been received.
    Ready { response: VersionedResponse, at: BlockNumber },
}
```

Listing 5: `QueryStatus` enumerate

[2]Callbacks are named `maybe_notify`.



(a) Current workflow (b) With timeout check

Figure 5.1: Logic flow for Pending requests

Timeout check on call to `maybe_notify` in `on_response`

`INFO1`	Check that `timeout` is inferior to the `current_block_number` in `on_response`		
Category	Informational		
Status	Present		
Rating	Severity: Info	Impact: None	Exploitability: None

The `timeout` specified in a `QueryStatus::Pending` is ignored in the `on_response` handler and comments of the pallet XCM `report_outcome` and `report_outcome_notify` document that `BlockNumber` as "the block number after which it is permissible for notify not to be called even if a response is received."

Thus it would be expected by a user of this API that the timeout might be respected, that is why an additionnal check on that timeout could be performed just after the match to check the `maybe_notify` presence. The `on_response` execution flow would be modified as displayed in green in Figure 5.1(b).

Check on the expected response type

`INFO2`	Add check on the expected response type		
Category	Informational		
Status	Present		
Rating	Severity: Info	Impact: None	Exploitability: None

```rust
pub enum Response {
    /// No response. Serves as a neutral default.
    Null,
    /// Some assets.
    Assets(MultiAssets),
    /// The outcome of an XCM instruction.
    ExecutionResult(Option<(u32, Error)>),
    /// An XCM version.
    Version(super::Version),
}
```

Listing 6: `Response` enumerate

`Response` is an enumeration composed of different types as shown in Listing 6. In the current design, the `Queries` storage stores the query id to make sure that an answer was previously requested with the correct `QueryStatus`. It could also be good practice to note whether this query expected a `Response::Assets` or a `Response::ExecutionResult`, for example. In the current situation, a request triggered by a `QueryHolding` could be theorically answered with a `Response::ExecutionResult`.

Handling of function `expecting_response`

`INFO3`	Implicit link between `expecting_response` and `AllowKnownQueryResponses`		
Category	Informational		
Status	Present		
Rating	Severity: Info	Impact: None	Exploitability: None

The `OnResponse` trait defines `expecting_response` and `on_response`. While the latter is called upon handling the `QueryResponse` instruction opcode, `expecting_response` is only called in the `AllowKnownQueryResponses` barrier. Kusama is configured to use this barrier so that function plays its role, but any parachain-template does not. Thus a parachain *forgetting* using that barrier could be misled thinking the function will be called by the executor. Putting the function in the barrier enables dropping early any unexpected `QueryResponse` but does not follow the same design choice that other functions defined by the config traits do which are, for most of them, called in instruction handlers.

A proposal is to remove the `expecting_response` call from the barrier and putting in the `QueryResponse` handler of `execute` in *polkadot/xcm/xcm-executor/src/lib.rs*, with a code similar to the one shown in Listing 7.

```
QueryResponse { query_id, response, max_weight } => {
    let origin = self.origin.as_ref().ok_or(XcmError::BadOrigin)?;
    if ResponseHandler::expecting_response(origin, query_id) {
        Config::ResponseHandler::on_response(origin, query_id, response,
↪   max_weight);
        Ok(())
    }
    else {
        Err(())
    }
}
```

Listing 7: QueryResponse instruction handler

It would make sure that the trait function is called regardless of the XCM barrier configuration and only postpone the check few calls later. A barrier can still be relevant to check that a message containing a `QueryResponse` instruction is only one instruction long.

5.9 AssetTrap and AssetClaims

The `AssetTrap` defines the action to perform on assets left in the holding of the XCVM after executing a message. These assets depending on the implementation can be claimed via the `ClaimAssets` trait implementation. The post [4] gives thorough explanations of the mechanism.

Both kusama and parachain-template use the default implementation provided by the XCM pallet which stores all traps in a storage. To be generic it does not store assets as a balance but instead, the storage key is the blake2 (256bits) of the `(origin, assets)` tuple. The value is the number of times this tuple has been trapped. To recover funds, a user has to claim assets of the exact assets trapped. Its origin is then used to perform the hash and the lookup in the storage which decrements the value or remove the entry. As such, the user has to perform as many claims as traps occured.

The security of this mechanism relies on the fact that the given assets can only be claimed by the same origin. The function properly decrements or removes the key from the map after a claim, preventing claiming the assets twice.

5.10 SubscriptionService

The trait defines the action to perform when receiving XCM version change notifications. One should implement a `start` and `stop` function. All configurations use the implementation of the XCM pallet. When receiving a `SubscribeVersion` the chain sends back an XCM message with its current version. The origin is properly checked and it also ensures the query has not already been performed. There are no impersonation nor version downgrade possibility in the current implementation.

6 XCM Executor

The XCVM is a register-based machine with a limited set of instructions. There are registers[1] such as `program`, `appendix`, `error_handler` that can modify the execution payload. Registers to count weight such as `surplus_weight` or `refunded_weight` are used to track the execution cost. Finally, the most important registers are `holding` which holds any assets manipulated by the execution and `origin` which hold the current context privileges (as a `MultiLocation`).

Instruction	Implem.	Origin(ok)	send XCM
`BuyExecution`	✓	-	-
`ClaimAsset`	✓	✓	-
`ClearError`	✓	-	-
`ClearOrigin`	✓	-	-
`DepositAsset`	✓	-	-
`DepositReserveAsset`	✓	-	✓
`DescendOrigin`	✓	✓	-
`ExchangeAsset`	✗	-	-
`HrmpNewChannelOpenRequest`	✗	-	-
`HrmpChannelAccepted`	✗	-	-
`HrmpChannelClosing`	✗	-	-
`InitiateReserveWithdraw`	✓	-	✓
`InitiateTeleport`	✓	-	✓
`QueryHolding`	✓	-	✓
`QueryResponse`	✓	✓	-
`ReceiveTeleportedAsset`	✓	✓	-
`RefundSurplus`	✓	-	-
`ReportError`	✓	-	✓
`ReserveAssetDeposited`	✓	✓	-
`SetErrorHandler`	✓	-	-
`SetAppendix`	✓	-	-
`SubscribeVersion`	✓	✓	-
`TransferAsset`	✓	✓	-
`TransferReserveAsset`	✓	✓	✓
`Transact`	✓	✓	-
`Trap`	✓	-	-
`UnsubscribeVersion`	✓	✓	-
`WithdrawAsset`	✓	✓	-

Implem: Implemented, Origin(ok): Origin is not None (no ClearOrigin before)

Table 6.1: XCM instructions opcodes

Table 6.1 shows all instruction mnemonics defined in the XCM executor implementation. Some of them are not implemented like HRMP channel negotiation (which can nonetheless be instantiated

[1] more precisely fields of a RUST

manually). The second column shows instructions that explicitly check origin not to be empty. Most of them will perform additional checks on the origin to make sure it is authorized to emit the instruction. The third column shows instructions that themselves emit new XCM messages (usually a response).

Message Loops and Recursion

In the current development state, XCM can not be led into creating dynamic recursion or infinite loop. Indeed, there are no loop construction inside the XCM "language" or its processing. Nevertheless, some instructions like `TransferReserveAsset`, `DepositReserveAsset`, `InitiateReserveWithdraw` and `InitiateTeleport` contain themselves, nested XCM messages.That's why modification of the handlers of such instructions could introduce infinite loops if they happen to insert one of those particular instructions in the message they sent. That is not the case, and in a general manner the XCM executor must not be modified. Note that the executor contains an unused constant about recursion.

```
/// The maximum recursion limit for `execute_xcm` and `execute_effects`.
pub const MAX_RECURSION_LIMIT: u32 = 8;
```

Nested messages in some instructions are not meant to be executed locally but sent to a destination. The only XCM instructions that could potentially trigger finite recursion is `Transact` by encoding the execution of an XCM containing a transact that is itself containing the execution of an XCM, etc.

On the relay chain side, in the configuration of `ump` pallet, `XcmSink` is used to decode the scale message from the queue and forward it to the executor (cf. *polkadot/runtime/parachains/src/ump.rs*). It implements the `process_upward_message` function that uses the scale decoder on `VersionedXcm`. In particular, it uses `decode_all_with_depth_limit` with the `xcm::MAX_XCM_DECODE_DEPTH` which is equal to 8. It means, for example, that a malicious XCM message containing finite recursive `Transact` cannot achieve more than 8 in depth or an error is returned at the scale decoding stage. The design is similar on the Parachain side, in cumulus. Parachains are using implementations from the `dmp-queue` and `xcmp-queue` pallets, or even the one of the `xcm` pallet in Cumulus repository that are using the same constant.

It is actually good practice to check for a depth limit while decoding potentially nested encoded data. It is important making such checks at this stage because the executor, by design, does not enforce a specific limit on nested message depth. Indeed, the executor only executes the message, instruction by instruction without analyzing the message as a whole so it's important that messages have been sanitized by the decoding step before.

6.1 Arithmetic Operations

The code is written in a very defensive manner. Most assets operations are performed with safe helper functions. For instance, to add assets, the executor uses the `subsume` function which depending on the underlying asset (fungible or non-fungible) properly adds with `saturating_add` or inserts the new non-fungible token.

Similarly, reducing the holding or the amount is also performed using helper functions like `checked_sub`, `saturating_sub` which prevents any underflow. Interestingly, some substractions remains with the pattern shown in Listing 8.

```
let weight = weight.min(self.0);
//[snip]
self.0 -= weight;
```

Listing 8: Arithmetic substraction protected by conditional check

Many arithmetic operations are performed in queue managers *ump.rs*, *dmp.rs* etc, but values are not directly controlled by an external user. It seems not possible to lead queues in overflowing.

6.2 Version Negotiation

The second blog post on XCM by Gavin Wood is a great introduction to version negotiation in the context of XCM [5]. Generally, it is supposed to be a transparent mechanism for users, which here are consensus systems like relay chains and parachains. The negotiation takes place in the background, alongside sending regular messages.

This process can be split into two steps, first trying to send a message and notice that we don't know the best version for the destination. It can be seen as lazy version negotiation. Secondly, it asks for the destination to send its supported version and handles the responses. The following subsections explain in detail how the protocols works, Figures 6.1 and 6.2 recap the two steps.

Sending a message and noting unknown version

To send a XCM message, a system needs to know in which version it can wrap its messages to make reception successful. For that, the XCM pallet has a storage named `SupportedVersion` in which it stores the supported version for destinations. Currently, the mechanism to fill that storage with this information starts at the sending stage of a message.

From a relay chain perspective, the XCM router finally used to send the message is located in *polkadot/runtime/common/src/xcm_sender.rs*. Before sending the message to the destination, or putting it in its dedicated queue in this case, it calls a function from the XCM pallet named `wrap_version(dest, msg)`.

This function checks the `SupportedVersion` storage and if the supported version for the specific destination is unknown, it calls the next function in the mechanism, `note_unknown_version(dest)`. Without interrupting the sending process, it will then wrap the message with the appropriate version if found. Otherwise, it uses the default one and returns it.

Function `note_unknown_version` uses another storage called `VersionDiscoveryQueue` which is "destinations whose latest XCM version we would like to know". It is a vector of bounded size, fixed in the configuration with the `VERSION_DISCOVERY_QUEUE_SIZE` storing those destinations and an integer specifying the number of times we needed that information. So the function iterates over that queue, bumps the integer if the location is already in the queue or else sets the destination with an integer to 1. Basically it notes that the version information needed to send the message was missing without interrupting the sending process. The process is summarized in Figure 6.1.

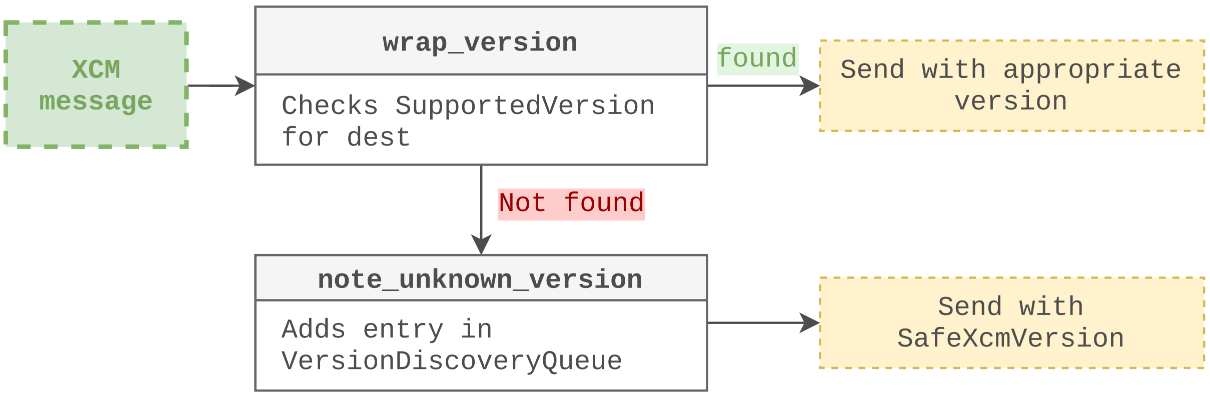

XCM message
wrap_version
Checks SupportedVersion for dest
found
Send with appropriate version
Not found
note_unknown_version
Adds entry in VersionDiscoveryQueue
Send with SafeXcmVersion

Figure 6.1: Sending message with appropriate version

Request a destination supported XCM version

Hooks. The XCM pallet hooks `on_initialize` [6] which is called at the initialization of every block. It can be considered as a routine that will be called at the creation of each block. The first part of that function is dedicated to version migration, the next part to version discovery. The function will get the `VersionDiscoveryQueue` mentioned in the previous section, trying to the most demanded destination. It will only retrieve one destination for which to query the XCM supported version. It will then call `request_version_notify(dest)` on that destination.

Function `request_version_notify(dest)` ensures that the destination was not already requested by checking the `VersionNotifiers` storage, precisely containing all locations that were already requested. It will then handle the query by incrementing the query ID, crafting a `SubscribeVersion` XCM message and sending it with `XcmRouter::send_xcm`. Finally, `VersionNotifiers` will be modified to include the new destination and the `Queries` storage, containing all the ongoing queries, will be also modified to add this query as a `VersionNotifier` type.

Thus this routine will eventually, at a maximum rate of one by block, request for version notifications from every system the chain has sent messages to.

Version request reception. From the receiver's perspective, version negotiation is the reception of the `SubscribeVersion` instruction. Upon receiving this instruction, the XCM executor calls `VersionChangeNotifier::start`. It first checks `VersionNotifyTargets` that stores all locations subscribed to the system XCM version change and the most recent associated version they were informed of. If it is not already subscribed, it then retrieves the version from `AdvertisedXcmVersion`, builds a `Response::Version` object to put in a `QueryResponse` XCM instruction and sends the message to the requestor. Finally, `VersionNotifyTargets` is mutated to take into account the new subscriber.

Note

There is also the UnsubscribeVersion instruction. On its reception the handler calls VersionChangeNotifier::stop that just removes the destination from VersionNotifyTarget.

Destination version response. Finally, the original sender that needed the version supported by the destination receives answers as QueryResponse XCM message. The handler in the executor only checks that the origin is not None and call ResponseHandler::on_response.

The on_response handler consists of a matcher on two patterns. In any case, it ensures the response is expected as it was registered beforehand in the Queries storage with a unique query ID. If not, it returns an UnexpectedResponse. Then it has a matcher for QueryStatus::VersionNotifier type of queries that will make sure that the origin that responded was the expected one, will mutate the state of the query to active and finally insert the answered version in SupportedVersion and then emit an event. The other matcher has already been addressed in Section 5.8.

Warning

As mentioned in some comments along the code, the handler always returns zero for the weight consumption.

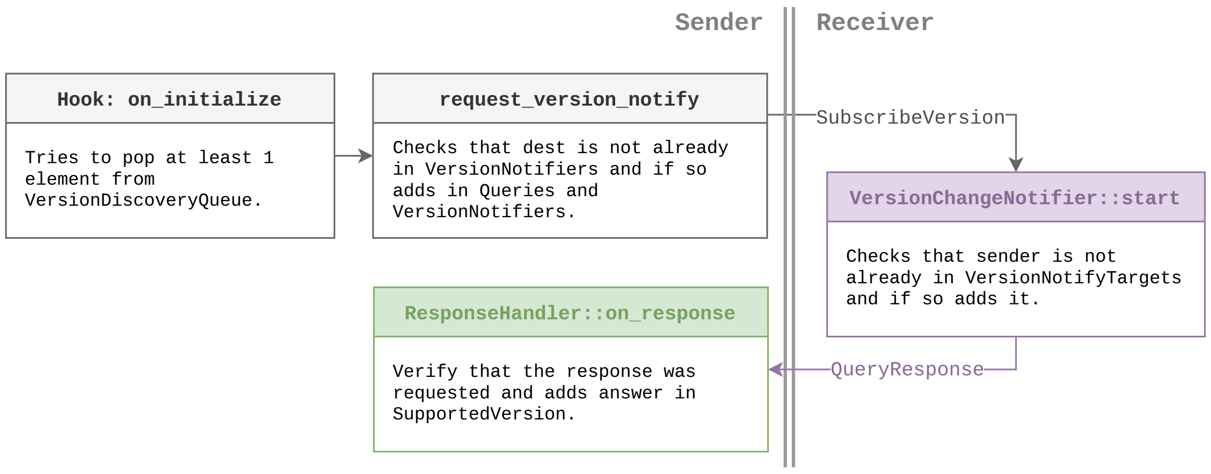

Sender
Receiver
Hook: on_initialize
Tries to pop at least 1 element from VersionDiscoveryQueue.
request_version_notify
Checks that dest is not already in VersionNotifiers and if so adds in Queries and VersionNotifiers.
SubscribeVersion
VersionChangeNotifier::start
Checks that sender is not already in VersionNotifyTargets and if so adds it.
QueryResponse
ResponseHandler::on_response
Verify that the response was requested and adds answer in SupportedVersion.

Figure 6.2: Version subscription

Conclusion. From a security perspective, the version negotiation is designed to allow systems to communicate by progressively learning their supported versions. It is a version discovery protocol and it seems that only the requested system can respond via verification of the query id and the origin. But nothing stops a system to send an invalid or an old-version XCM message. It is then the responsibility of the various XcmSink implementations to filter or convert unsupported version.

Note

In the `XcmSink` implementation of the UMP queue used by the relay chains, at *polkadot/runtime/parachains/src/ump.rs* line 103, the `process_upward_message` function will try to convert old XCM versions into XCM v2 messages just after decoding the message. Some deprecated instructions are no longer supported or cannot be converted and this step will then throw an error.

6.3 Dynamic Testing

As part of the audit, multiple dynamic tests have been performed to assess the behavior of the relay chains and parachain on some tests. While it would have been possible to use the XCM simulator[2] or to write tests directly in RUST, most tests have been performed using a black-box approach ensuring tests represent an action that an attack can perform on a running "mainnet" chain.

While Javascript APIs provides all base utilities to connect to a node and submit extrinsics, Quarkslab chose the great `py-substrate-interface`[3] Python framework also based on `py-scale-codec`[4] that enables serializing/deserializing SCALE [7] messages. A thin wrapper has been developed on these tools to interact easily with a node within a Python shell. Thin wrappers have also been written to craft XCM messages easily and send them to a chain. Listing 9 shows a script to perform a reserve transfer.

[2]https://github.com/paritytech/polkadot/tree/master/xcm/xcm-simulator
[3]https://github.com/polkascan/py-substrate-interface
[4]https://github.com/polkascan/py-scale-codec

```
from subshell import SubstrateNode

# Connect to the relay chain and set an identity to sign extrinsics automatically
relay = SubstrateNode.from_uri("ws://localhost:9944", root_xcm=True)
relay.set_identity(Keypair.create_from_uri("//Alice"))

from subshell.xcm import *

# Connect to the parachain
parachain = SubstrateNode.from_uri("ws://localhost:9988")

def listen_para_blocks(block, ith, id):
    for ex in block.extrinsics:  # iterate extrinsics of the block
        if ex.module == "ParachainSystem" and ex.function ==
        ↪  "set_validation_data":
            # check events raised which might contains a dmpqueue event
            for evt in ex.events:
                if evt.pallet_name == "DmpQueue" and evt.name ==
                ↪  "ExecutedDownward":
                    outcome = evt.args[1]
                    print(f"[parachain] XCM execution result:
                    ↪  {list(outcome.keys())}")
                    return "" # stop listening

# register block listener on parachain
parachain.subscribe_block_non_blocking(listen_para_blocks)

# Craft assets and multilocation on relay chain side
parents = 0
amount = 1.
beneficiary = "0x" + Keypair.create_from_uri("//Eve").public_key.hex()

dest = VersionedMultiLocation(MultiLocation(parents, Junctions(Parachain(1000))))
benef = VersionedMultiLocation(
            MultiLocation(parents,
                  Junctions(AccountId32(Any(), beneficiary))))

assets = VersionedMultiAssets(
    MultiAssets(
        MultiAsset(Concrete(MultiLocation(parents, Junctions.Here())),
        ↪  Fungible(amount))
    ))

# Submit extrinsic transaction from relay chain with Alice account
res = relay.xcmpallet.limited_reserve_transfer_assets(dest, benef, assets, 0,
↪  Unlimited(), wait_inclusion=True)

# iterate events to check if it has been submitted
for evt in res.events:
    if evt.pallet_name == "XcmPallet" and evt.name == "Attempted":
        outcome = evt.args
        print(f"[relay-chain] XCM execution result: {list(outcome.keys())}")
```

Listing 9: Limit reserve transfer asset script

The script connects to both chains, crafts the reserve transfer arguments and submits the extrinsic. It then checks on both chains that the message has properly been submitted on one side and executed on the other. Indeed, from a single side like relay chain it is not possible to know if a message have properly been successful on the other-side as there is no information back from the parachain to the relay chain. Few scenarios have been tested in such a black-box manner to verify that chains behaves as static analysis review suggests.

Fuzzing

An elementary fuzzer have been developed to fuzz any extrinsic with random values, yet valid with regards to scale codec. Despite being functional, the randomness nature of mutations makes it difficult to generate interesting test-cases. Its application on XCM extrinsics appeared to be very inefficient because each extrinsic is called sequentially and because barriers are very restrictive. Indeed barriers are very effective at rejecting any malformed XCM messages. Getting further in this research would require generating more relevant messages with meaningful weights, accounts, etc. Very little time has been dedicated to fuzzing and cannot be considered as conclusive.

Note

Slightly invalid scale encoded messages have also been submitted as extrinsic calls. They were all rejected by the transaction validation function. As this part is out-of-scope, only valid SCALE messages have been tested.

6.4 Origin and execution privileges

XCM Origin vs Runtime Origin

XCM origin. The XCM privilege model relies heavily on message origin (MultiLocation). There are some instructions that can only be executed when the origin is set to certain values, for example, the relay chain itself instead of a specific account (cf. Table 6.1 for instructions that check that origin is different than `None`). In the specification of the Polkadot Cross-Consensus Message (XCM) Format [8], the origin registry of the XCM virtual machine is described as:

> *Expresses the location with whose authority the current program is running. May be reset to None at will (implying no authority), and may also be set to a strictly interior location at will (implying a strict subset of authority).*

In the case of sending from the relay chain to a parachain, the origin of the message, for the parachain, will always be the relay chain itself. Although the origin register cannot be directly altered, two instructions, `ClearOrigin` and `DescendOrigin` can respectively reset the registry to `None` or set it to a strictly interior location. For example, to downgrade the origin from the relay chain to a local account that signed an extrinsic to emit this message.

Runtime origin. The substrate documentation [9] explains that "[t]he runtime origin is used by dispatchable functions to check where a call has come from". Indeed, every callable extrinsic need to have `origin: OriginFor<T>` as their first argument, and this parameter will be implicitly filled at runtime when making an extrinsic call.

Note

Runtime and XCM origins are two different notions. XCM configuration traits should define how converting an XCM origin (MultiLocation) to a runtime origin and vice-versa. It will be needed, for example, for `Transact` that needs a runtime origin from an XCM origin to make an extrinsic call.

Conclusion. The privileges execution model inside XCM relies on the content of the origin registry during execution. This registry is always set with the literal origin of the message. But to avoid that every message executes with the privileges of the direct sender, which is the sending system itself, some instructions can be used to drop the origin or to downgrade to a lower privilege origin.

Origin in the context of `pallet_xcm`

In the context of relay chain to parachain communication, Figure 6.3 gives an overview of the call path to `send_xcm` of the `XcmRouter`, which is the last step before the message is deposited into queues and then picked up by parachains. There are three ways to get to that method from the `pallet_xcm`:

1. The `send` extrinsic, used to send an arbitrary XCM message. It will use a local `send_xcm` method that will prepend `DescendOrigin(interior)` to the message for the receiver to be aware of the sender of the message, which is the account that signed the extrinsic call `send`.
2. The `force_subscribe_version_notify` and `force_unsubscribe_version_notify` extrinsics that must be called as root that will send a static XCM message with the chain as origin.
3. Particular XCM instructions, such as `TransferReserveAsset` for example, or the others referenced in the Table 6.1 as "Sending XCM", because they will effectively send a new XCM message when executed. These special instructions are further investigated in following Section 6.4 because they will continue the execution on the destination with the sender chain authority as origin.

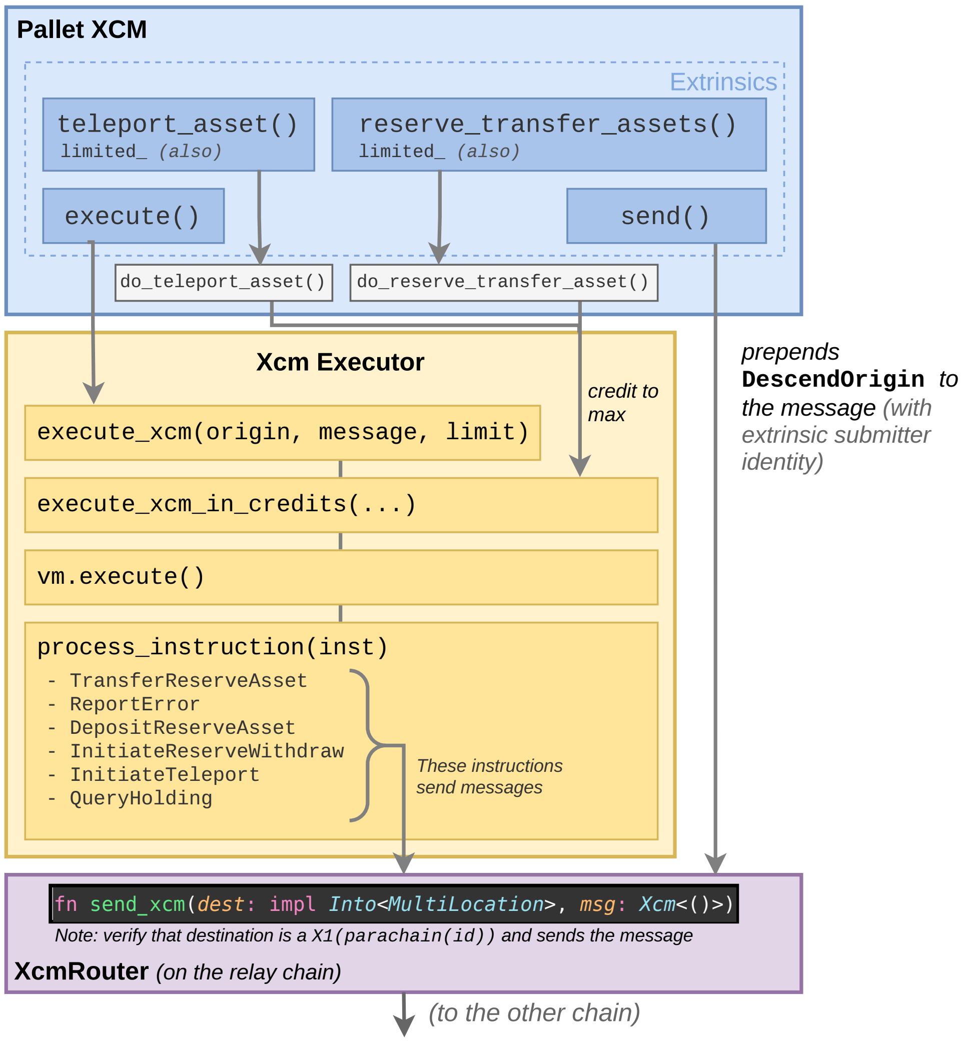

Pallet XCM
Extrinsics
teleport_asset()
limited_ (also)
reserve_transfer_assets()
limited_ (also)
execute()
send()
do_teleport_asset()
do_reserve_transfer_asset()
Xcm Executor
credit to max
execute_xcm(origin, message, limit)
execute_xcm_in_credits(...)
vm.execute()
process_instruction(inst)
- TransferReserveAsset
- ReportError
- DepositReserveAsset
- InitiateReserveWithdraw
- InitiateTeleport
- QueryHolding
These instructions send messages
prepends DescendOrigin to the message (with extrinsic submitter identity)
fn send_xcm(dest: impl Into<MultiLocation>, msg: Xcm<()>)
Note: verify that destination is a X1(parachain(id)) and sends the message
XcmRouter (on the relay chain)
(to the other chain)

Figure 6.3: Usage of send xcm

Instruction sending XCM message with high privileges origin

Escalating privileges from an account would imply being able to write an arbitrary message that will be executed with a “superior” authority as origin, such as the relay chain itself.

With the `send` extrinsic, escalation is not possible because a `DescendOrigin(interior)` will be prepended as the first instruction of the message, thus lowering privileges immediately. The `force_subscribe_version_notify` and `force_unsubscribe_version_notify` bypass the privileges-drop mechanism but require to be root and send static message anyway. Then `teleport_assets` and `reserve_transfer_assets` or the associated `limited` also send message with relay chain as origin but use static messages. The last extrinsic that can be used is `execute`, that can contain an arbitrary message, and thus use instruction that will themselves send XCM messages on behalf of the relay chain. Let’s see what messages can be sent via these instructions.

- **`TransferReserveAsset`**: this instruction takes an XCM message as its third argument. But the content of the message will be appended to a static one containing `ClearOrigin` as its last instruction. Thus all custom input will be executed with origin as `None`.

```
let mut message = vec![ReserveAssetDeposited(assets), ClearOrigin];
message.extend(xcm.0.into_iter());
```

- **`DepositReserveAsset`**: it sends the same XCM message as `TransferReserveAsset`.
- **`InitiateReserveWithdraw`**: it sends almost the same XCM message as `TransferReserve-Asset` but with a different first instruction.

```
let mut message = vec![WithdrawAsset(assets), ClearOrigin];
message.extend(xcm.0.into_iter());
```

- **`InitiateTeleport`**: it sends almost the same XCM message as `TransferReserveAsset` but with a different first instruction.

```
let mut message = vec![ReceiveTeleportedAsset(assets), ClearOrigin];
message.extend(xcm.0.into_iter());
```

- **`ReportError`**: it sends a static XCM message with the content of the error registry as response.

```
let response = Response::ExecutionResult(self.error);
let message = QueryResponse { query_id, response, max_weight };
```

- **`QueryHolding`**: it sends the same XCM message as `ReportError` but with `assets` from the holding registry as response.

To conclude, it is not possible for a user to use the `pallet_xcm` to send a message with the relay chain authority as origin.

Warning

Any modifications to the executor codebase have to be done very carefully. The privilege downgrade of the sender system is entirely its responsibility because the receiver processes all messages with the sender as origin before any execution. Then, forgetting any `ClearOrigin` instructions in the injection of the embedded messages shown in the previous list will lead to executing arbitrary code on a destination system with the origin of the sender system.

7 Asset Transfer

Being the main use-case of XCM, transferring assets is the most important scenario. The XCM format enables two main kind of asset transfers behaving differently. These two are teleporting and reserving.

Parity Tech provides some documentation on these asset transfer mechanisms [2, 10]. In this context, it is important to check that assets are properly locked on one chain before being released on the second one. For teleported, assets are burned on one side and minted on the destination side. **The security model relies on the trust between chains exchanging the assets.** Indeed, there is, for now, no automatic mechanism for a chain to ensure that assets have been properly locked on the other side before releasing them locally.

These operations can be performed between the relay chain and a parachain but also between parachains. The inner working (extrinsics used, queues, etc.) will differ (see Appendix A) but from a functional perspective it will work the same way, depending on the chain configuration.

7.1 Withdraw and Deposit

`WithdrawAsset` and `DepositAsset` are "primitive" instructions to respectively remove on-chain assets to the XCVM holding registry, and conversely remove assets from the XCVM holding registry adding equivalent assets on-chain. Listing 10 and 11 show respectively `WithdrawAsset` and `DepositAsset` handlers in XCM executor.

```
WithdrawAsset(assets) => {
    // Take `assets` from the origin account (on-chain) and place in holding.
    let origin = self.origin.as_ref().ok_or(XcmError::BadOrigin)?;
    for asset in assets.drain().into_iter() {
        Config::AssetTransactor::withdraw_asset(&asset, origin)?;
        self.holding.subsume(asset);
    }
    Ok(())
}
```

Listing 10: `WithdrawAsset` XCM executor handler

```
DepositAsset { assets, max_assets, beneficiary } => {
    let deposited = self.holding.limited_saturating_take(assets, max_assets as
↪   usize);
    for asset in deposited.into_assets_iter() {
        Config::AssetTransactor::deposit_asset(&asset, &beneficiary)?;
    }
    Ok(())
}
```

Listing 11: `DepositAsset` XCM executor handler

It should be made very clear that with these two instructions alone, it is only possible to make **local transfers**, with the `execute` extrinsic, for example, crafting a `[WithdrawAsset, DepositAsset]` to send some assets to a local beneficiary. To make cross-chain transfers, other instructions such as `InitiateTeleport` or `TransferReserveAsset` are needed and described in next sections.

7.2 Teleport and Reserve Transfers

To move assets across chains via XCM, as explained in Section 4.1, one must use `teleport_assets`, `reserve_transfer_assets` extrinsics or their `limited` variants. These instructions execute an XCM message locally and then send a message to the desired destination to make the transfer effective. Additional checks are specified since this part of the execution, locally and remotely, has to be done in the name (Origin) of the sender system, instead of the user that initiated it. That is why a `ClearOrigin` will be injected in the sent message to prevent any following custom instructions being interpreted with a wrong origin.

Teleport Assets

Calling `teleport_assets` or `limited_teleport_assets`, the executed message will be composed of a `WithdrawAsset` and a `InitiateTeleport` instruction. `WithdrawAsset` is straightforward and will take some assets from the origin account and put it in the XCVM holding registry. The `InitiateTeleport` instruction in Listing 12 should be investigated a little bit further:

```
InitiateTeleport { assets, dest, xcm } => {
    // We must do this first in order to resolve wildcards.
    let assets = self.holding.saturating_take(assets);
    for asset in assets.assets_iter() {
        Config::AssetTransactor::check_out(&dest, &asset);
    }
    let assets = Self::reanchored(assets, &dest)?;
    let mut message = vec![ReceiveTeleportedAsset(assets), ClearOrigin];
    message.extend(xcm.0.into_iter());
    Config::XcmSender::send_xcm(dest, Xcm(message)).map_err(Into::into)
}
```

Listing 12: `InitiateTeleport` XCM executor handler

```
ReceiveTeleportedAsset(assets) => {
    let origin = self.origin.as_ref().ok_or(XcmError::BadOrigin)?;
    // check whether we trust origin to teleport this asset to us via config
↪   trait.
    for asset in assets.inner() {
        // We only trust the origin to send us assets that they identify as their
        // sovereign assets.
        ensure!(
            Config::IsTeleporter::filter_asset_location(asset, origin),
            XcmError::UntrustedTeleportLocation
        );
        // We should check that the asset can actually be teleported in (for this
↪   to be in error, there
        // would need to be an accounting violation by one of the trusted chains,
↪   so it's unlikely, but we
        // don't want to punish a possibly innocent chain/user).
        Config::AssetTransactor::can_check_in(&origin, asset)?;
    }
    for asset in assets.drain().into_iter() {
        Config::AssetTransactor::check_in(origin, &asset);
        self.holding.subsume(asset);
    }
    Ok(())
}
```

Listing 13: `ReceiveTeleportedAsset` XCM executor handler

The instruction has the following behavior:

- First, assets specified in parameters are taken from the holding registry with `self.holding.saturating_take(assets)`.
- Then, `AssetTransactor::check_out` is called, it will generally increase the assets in a special "teleported" account, it is bookkeeping.
- The `reanchored` function is called on the assets with the destination to invert the relative location.
- Two instructions, `[ReceiveTeleportedAsset(assets), ClearOrigin]` are inserted at the beginning of the message that will be sent to the destination.
- The embedded modified message is sent to the destination.

Upon reception, the destination will then execute `ReceiveTeleportedAsset` shown in Listing 13 first.

Here is a break-down of this instruction:

- First, the origin is copied from the origin registry and must be different from `none` otherwise a `BadOrigin` is returned.
- Then, `IsTeleporter` is called on the assets. On statemine, it's a function that only accepts native assets.

- Next, the AssetTransactor::can_check_in and AssetTransactor::check_in will be called to note and remove the asset from the bookkeeping account. Statemine uses a tuple for AssetTransactor composed of (CurrencyTransactor, FungiblesTransactor), the first one is the same as the one used on Kusama, to handle native coin, and the second is specific to Statemine, to handle other fungible assets. Their implementations are quite similar except one is using the Currency trait and the other one the Asset trait. The check_in method will respectively implement the withdraw/burn of the asset on the bookkeeping account.
- Finally, self.holding.subsume(asset) adds the assets in parameters to the holding registry, ready to be deposited by the next instructions.

In short, a teleport can somehow be summarized to a self.holding.saturating_take in the sender XCVM, and a self.holding.subsume in the destination XCVM. Since the assets are not directly stored anywhere in between, except with the bookkeeping that keeps a trace of the total amounts teleported, assets might be lost if the ReceiveTeleportedAsset is never executed correctly for any reason. For a security perspective there is no mean of teleporting assets if the remote chain has not explicitly been granted to do so in the configuration.

Reserve Assets

The reserve_transfer_assets and limit_reserve_transfer_assets are quite similar to the teleport ones. One difference is that the do_reserve_transfer_assets function creates a message with only a TransferReserveAsset instruction without a WithdrawAsset first. It's because the TransferReserveAsset instruction performs an AssetTransactor::beam_asset call that will transfer the assets from the account of the origin of the message to the reserve account associated with the destination.

```
TransferReserveAsset { mut assets, dest, xcm } => {
    let origin = self.origin.as_ref().ok_or(XcmError::BadOrigin)?;
    // Take `assets` from the origin account (on-chain) and place into dest
↪   account.
    let inv_dest = Config::LocationInverter::invert_location(&dest)
        .map_err(|()| XcmError::MultiLocationNotInvertible)?;
    for asset in assets.inner() {
        Config::AssetTransactor::beam_asset(asset, origin, &dest)?;
    }
    assets.reanchor(&inv_dest).map_err(|()| XcmError::MultiLocationFull)?;
    let mut message = vec![ReserveAssetDeposited(assets), ClearOrigin];
    message.extend(xcm.0.into_iter());
    Config::XcmSender::send_xcm(dest, Xcm(message)).map_err(Into::into)
}
```

Step by step, this execution of this instruction:

- First, the origin is retrieved from the origin registry, and a None origin will return a BadOrigin.
- Then, the destination location is inverted for the future call of assets.reanchor.

- Then, an `AssetTransactor::beam_asset` is performed to transfer the asset from the sender (origin) to the local destination reserve account.
- Finally, `[ReserveAssetDeposited(assets), ClearOrigin]` instructions are inserted at the beginning of nested XCM message and everything is sent to the destination.

Then the destination receives the message and executes the first instruction `ReserveAssetDeposited` as shown in Listing 14:

```
ReserveAssetDeposited(assets) => {
    // check whether we trust origin to be our reserve location for this asset.
    let origin = self.origin.as_ref().ok_or(XcmError::BadOrigin)?;
    for asset in assets.drain().into_iter() {
        // Must ensure that we recognise the asset as being managed by the origin.
        ensure!(
            Config::IsReserve::filter_asset_location(&asset, origin),
            XcmError::UntrustedReserveLocation
        );
        self.holding.subsume(asset);
    }
    Ok(())
}
```

Listing 14: `ReserveAssetDeposited` XCM executor handler

The execution of this instruction is straightforward, it checks that the origin and the asset satisfy the `IsReserve` filter and then calls `self.holding.subsume(asset)` that adds the assets in the holding registry. The following instruction `ClearOrigin`, as usual, will reset the origin registry to `None`, `BuyExecution` will pay for the execution and `DepositAsset` will retrieve the asset from the registry to the account of a beneficiary.

Note

`TransferReserveAsset`, differently from `InitiateTeleport`, directly does the transfer from the origin to the reserve account. But an unused instruction, in the Polkadot codebase, `DepositReserveAsset` performs a very similar action than `TransferReserveAsset` beside taking the asset from the holding registry, like `InitiateTeleport`.

It is worth noticing that another instruction, unused in the codebase, `InitiateReserveWithdraw` is the counterpart of the `TransferReserveAsset` or `DepositReserveAsset` to retrieve the asset that was reserved in a special account on the sender side. It behaves similarly to those other instructions, injecting two instructions `[WithdrawAsset(assets), ClearOrigin]` at the beginning of the sent message, that will be executed with the sending system as origin, and thus withdraw from the corresponding reserve account.

To sum up, a `reserve` can somehow be summarized to a transfer to a reserve account on the sender chain, and a `self.holding.subsume` in the destination XCVM. Since the assets are reserved in the account, `InitiateReserveWithdraw` makes the inverted path possible to get back the assets on the initial chain. The current implementation does not raise any particular issue.

8 Conclusion

The cornerstone of XCM security is a very careful handling of the origin (*MultiLocation*) in the XCM pallet and executor. Any bad configuration or missing check might introduce vulnerabilities allowing unwanted transfers, etc. Thankfully, the audit did not reveal any misconception on this aspect.

However, even though proposed configurations are coherent as a whole, any careless change in a configuration trait behavior might have security consequences. Any parachain developer shall be extremely cautious changing default configurations (cf. [3]). Indeed, there are implicit dependencies between XCM traits. For example, presence or not of the `QueryResponse` barrier have direct impact on the execution or not of the `expecting_response` of the `ResponseHandler` config (cf. 5.8). As another example, the handling of `BuyExecution` instruction heavily relies on the presence of the `AllowTopLevelPaidExecutionFrom` barrier.

Asset transfer security (reserves and teleports) is not trustless. It is critically important to trust the chain before accepting it as a reserve or for teleports. Indeed, there are currently no mechanisms during transfers ensuring on the receiving side that assets have properly been locked or burnt on the sending side. It also trusts the sending chain to have properly set a `ClearOrigin` for these operations. As such, two chains exchanging assets are security-wise morally interdependent.

Finally, the default XCM configuration is very conservative by default (denying all) and is deemed secure. Quarkslab did not revealed additional[1] fairness, or DOS issues.

XCM security boils down to deciding who is trusted as a reserve or a teleport origin and what to allow to be executed thanks the various barriers, filters and origin checks. No inherent significant issues have been found thanks to the XCM well-thought-out design and implementation.

[1]At the time of the audit already acknowledged issues were in the process of being fixed

Glossary

collator A node that maintains a parachain by collecting parachain transactions and producing state transition proofs for the validators.

Cross-Consensus Messaging Common Messaging Format for information, asset transfer between substrate-based chains.

Cross-Consensus Messaging Protocol Main protocol allowing exchanging XCM messages. The protocol features version negotation, channel opening negotiation and various queues management.

Cross-Consensus Virtual Machine The XCVM is a register-based machine, none of whose registers are general purpose. The XCVM instruction format, set of machine registers and definitions of interaction therefore compromise the bulk of the XCM message.

extrinsic An extrinsic is a piece of information that comes from outside the chain and is included in a block. Extrinsics fall into three categories: inherents, signed transactions, and unsigned transactions.

pallet Substrate modules exposing various extrinsics, events, errors and storage items that will be compiled in the runtime and usable by users or other components. It is implemented as RUST crates.

validator A node that secures the Relay Chain by staking DOT, validating proofs from collators on parachains and voting on consensus along with other validators.

Acronyms

DMP	Downward Message Passing.
HRMP	HoRizontal Message Passing.
PC-PC	Parachain-Parachain.
PoS	Proof-Of-Stake.
RC-PC	Relaychain-Parachain.
UMP	Upward Message Passing.
XCM	Cross-Consensus Messaging.
XCMP	Cross-Consensus Messaging Protocol.
XCVM	Cross-Consensus Virtual Machine.

Bibliography

[1] Web3 Foundation. *XCMP Overview*. Dec. 27, 2021. URL: https://research.web3.foundation/en/latest/polkadot/XCMP/index.html (visited on Dec. 27, 2021) (cit. on p. 4).

[2] Gavin Wood. *XCM: The Cross-Consensus Message Format*. Sept. 6, 2021. URL: https://medium.com/polkadot-network/xcm-the-cross-consensus-message-format-3b77b1373392 (visited on Dec. 27, 2021) (cit. on pp. 4, 12, 35).

[3] *Kusama's Governance Thwarts Would-be Attacker!* Oct. 19, 2021. URL: https://medium.com/kusama-network/kusamas-governance-thwarts-would-be-attacker-9023180f6fb (visited on Dec. 28, 2021) (cit. on pp. 11, 41).

[4] Gavin Wood. *XCM Part III: Execution and Error Management*. Sept. 29, 2021. URL: https://medium.com/polkadot-network/xcm-part-iii-execution-and-error-management-ceb8155dd166 (visited on Dec. 27, 2021) (cit. on p. 23).

[5] Gavin Wood. *XCM Part II: Versioning and Compatibility*. Sept. 15, 2021. URL: https://medium.com/polkadot-network/xcm-part-ii-versioning-and-compatibility-b313fc257b83 (visited on Dec. 27, 2021) (cit. on p. 26).

[6] *Documentation on the Hooks trait*. on_initialize. URL: https://docs.substrate.io/rustdocs/latest/frame_support/traits/trait.Hooks.html#method.on_initialize (visited on Jan. 7, 2022) (cit. on p. 27).

[7] *SCALE Codec*. Version 3.0. Dec. 15, 2021. URL: https://docs.substrate.io/v3/advanced/scale-codec/ (visited on Dec. 28, 2021) (cit. on p. 29).

[8] Gavin Wood. *Polkadot Cross-Consensus Message (XCM) Format*. Version Version 2, final. Oct. 17, 2021. URL: https://github.com/paritytech/xcm-format/tree/fefa511687469c9a34759465d88e3b07e3ed6d22 (visited on Dec. 28, 2021) (cit. on p. 31).

[9] *Origins*. Version 3.0. Dec. 15, 2021. URL: https://docs.substrate.io/v3/runtime/origins/ (visited on Dec. 28, 2021) (cit. on p. 32).

[10] Shawn Tabrizi. *XCM Workshop*. Oct. 19, 2021. URL: http://www.shawntabrizi.com/xcm-workshop/#/ (visited on Dec. 28, 2021) (cit. on p. 35).

Appendix A

Message Types

The polkadot framework defines three types of messages that can be exchanged. These messages are the following:

- Downward Message Passing (DMP): From Relay-chain to Parachains
- Upward Message Passing (UMP): From Parachains to Relay-chain
- HoRizontal Message Passing (HRMP): From Parachains to Parachains

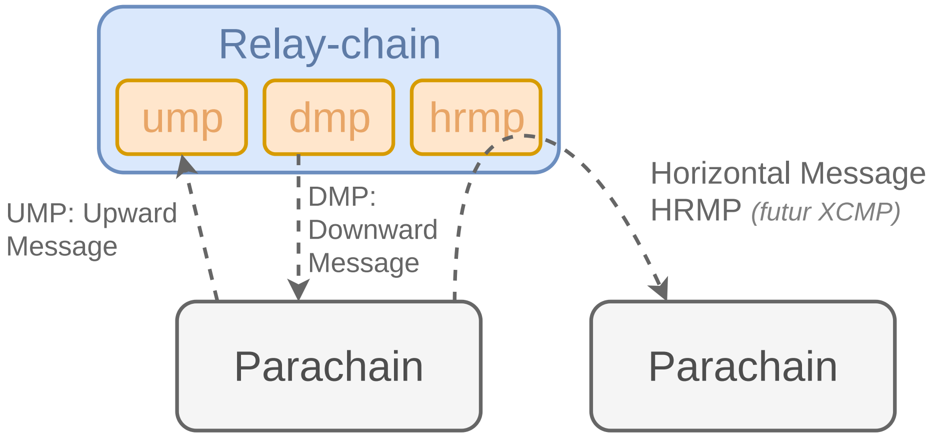


Figure A.1: Polkadot message types

As shown on Figure A.1, some pallets handle messages on relay-chain. As such, UMP and DMP messages are temporarily stored on the relay-chain before being processed or dispatched to the parachain. At the time of writing[1], parachain-to-parachain is implemented through HRMP and thus have to transit via the relay-chain. In the futur PC-to-PC (Cross-Consensus Messaging Protocol (XCMP)) communications are expected to be performed via direct communication between parachains.

[1]December 26th, 2021

Appendix B

XCM Types

This appendix shows basic XCM types used and their differences from version to version.

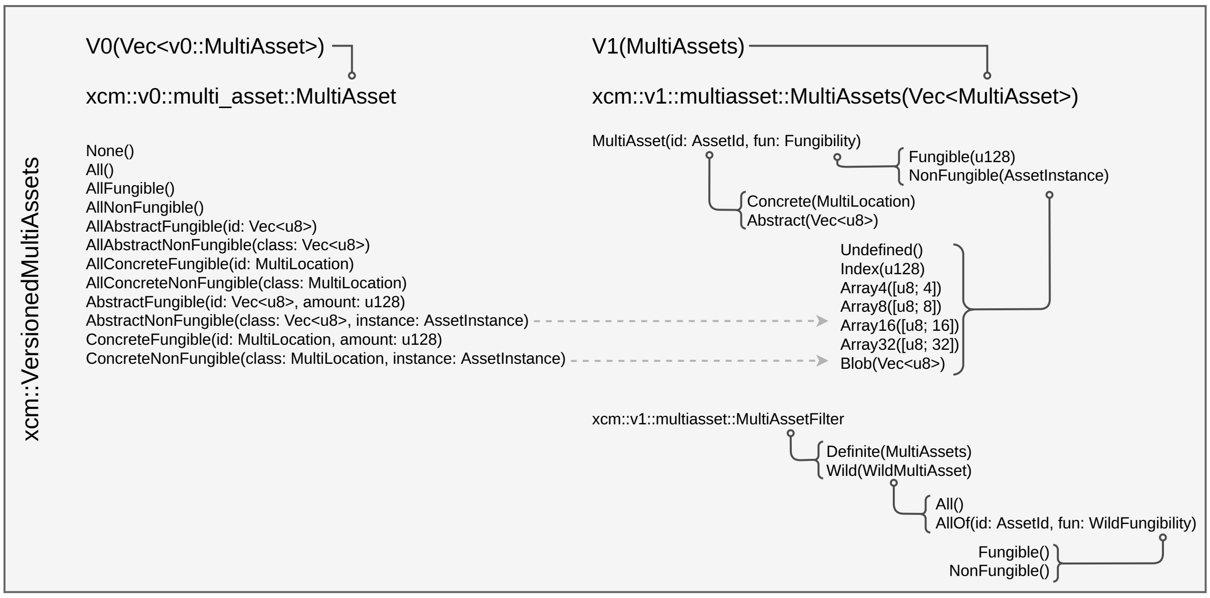

xcm::VersionedMultiAssets
V0(Vec<v0::MultiAsset>)
xcm::v0::multi_asset::MultiAsset
None()
All()
AllFungible()
AllNonFungible()
AllAbstractFungible(id: Vec<u8>)
AllAbstractNonFungible(class: Vec<u8>)
AllConcreteFungible(id: MultiLocation)
AllConcreteNonFungible(class: MultiLocation)
AbstractFungible(id: Vec<u8>, amount: u128)
AbstractNonFungible(class: Vec<u8>, instance: AssetInstance)
ConcreteFungible(id: MultiLocation, amount: u128)
ConcreteNonFungible(class: MultiLocation, instance: AssetInstance)
V1(MultiAssets)
xcm::v1::multiasset::MultiAssets(Vec<MultiAsset>)
MultiAsset(id: AssetId, fun: Fungibility)
Fungible(u128)
NonFungible(AssetInstance)
Concrete(MultiLocation)
Abstract(Vec<u8>)
Undefined()
Index(u128)
Array4([u8; 4])
Array8([u8; 8])
Array16([u8; 16])
Array32([u8; 32])
Blob(Vec<u8>)
xcm::v1::multiasset::MultiAssetFilter
Definite(MultiAssets)
Wild(WildMultiAsset)
All()
AllOf(id: AssetId, fun: WildFungibility)
Fungible()
NonFungible()

Figure B.1: XCM Assets

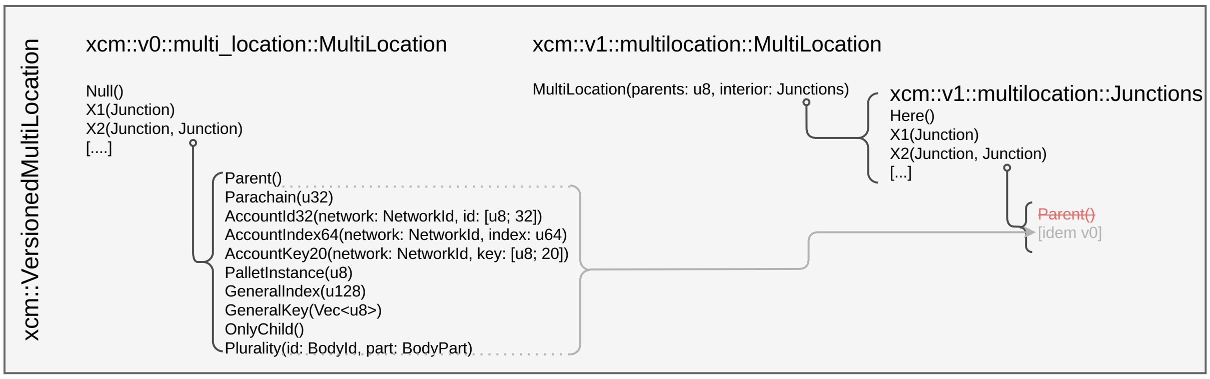
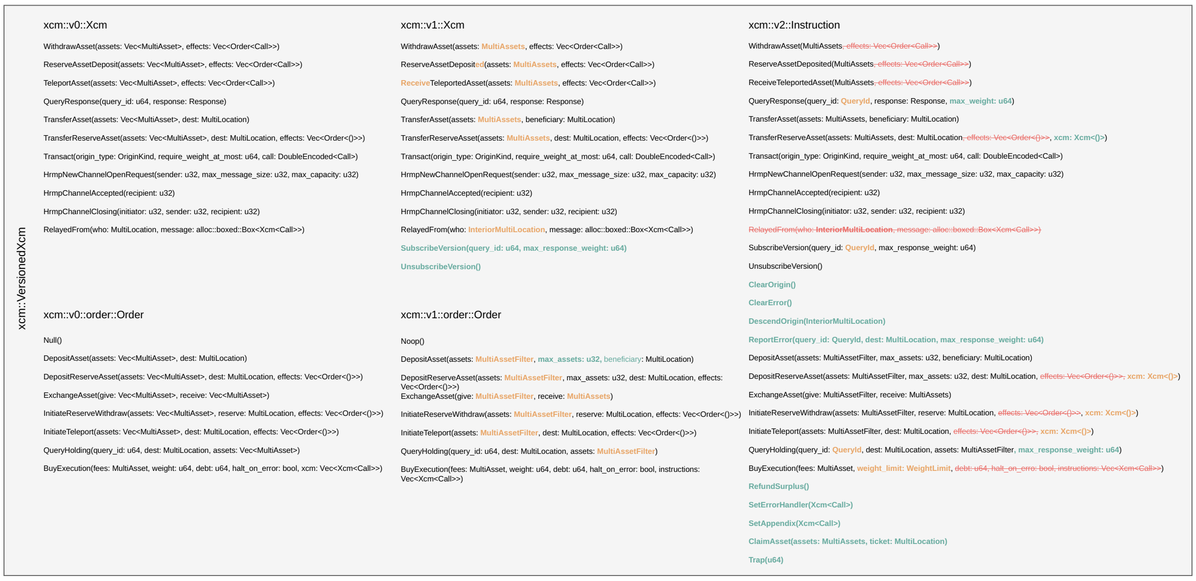

xcm::VersionedMultiLocation
xcm::v0::multi_location::MultiLocation
Null()
X1(Junction)
X2(Junction, Junction)
[....]
Parent()
Parachain(u32)
AccountId32(network: NetworkId, id: [u8; 32])
AccountIndex64(network: NetworkId, index: u64)
AccountKey20(network: NetworkId, key: [u8; 20])
PalletInstance(u8)
GeneralIndex(u128)
GeneralKey(Vec<u8>)
OnlyChild()
Plurality(id: BodyId, part: BodyPart)
xcm::v1::multilocation::MultiLocation
MultiLocation(parents: u8, interior: Junctions)
xcm::v1::multilocation::Junctions
Here()
X1(Junction)
X2(Junction, Junction)
[...]
~~Parent()~~
[idem v0]

Figure B.2: XCM Multilocations



xcm::VersionedXcm

xcm::v0::Xcm

WithdrawAsset(assets: Vec<MultiAsset>, effects: Vec<Order<Call>>)
ReserveAssetDeposit(assets: Vec<MultiAsset>, effects: Vec<Order<Call>>)
TeleportAsset(assets: Vec<MultiAsset>, effects: Vec<Order<Call>>)
QueryResponse(query_id: u64, response: Response)
TransferAsset(assets: Vec<MultiAsset>, dest: MultiLocation)
TransferReserveAsset(assets: Vec<MultiAsset>, dest: MultiLocation, effects: Vec<Order<()>>)
Transact(origin_type: OriginKind, require_weight_at_most: u64, call: DoubleEncoded<Call>)
HrmpNewChannelOpenRequest(sender: u32, max_message_size: u32, max_capacity: u32)
HrmpChannelAccepted(recipient: u32)
HrmpChannelClosing(initiator: u32, sender: u32, recipient: u32)
RelayedFrom(who: MultiLocation, message: alloc::boxed::Box<Xcm<Call>>)

xcm::v0::order::Order

Null()
DepositAsset(assets: Vec<MultiAsset>, dest: MultiLocation)
DepositReserveAsset(assets: Vec<MultiAsset>, dest: MultiLocation, effects: Vec<Order<()>>)
ExchangeAsset(give: Vec<MultiAsset>, receive: Vec<MultiAsset>)
InitiateReserveWithdraw(assets: Vec<MultiAsset>, reserve: MultiLocation, effects: Vec<Order<()>>)
InitiateTeleport(assets: Vec<MultiAsset>, dest: MultiLocation, effects: Vec<Order<()>>)
QueryHolding(query_id: u64, dest: MultiLocation, assets: Vec<MultiAsset>)
BuyExecution(fees: MultiAsset, weight: u64, debt: u64, halt_on_error: bool, xcm: Vec<Xcm<Call>>)

xcm::v1::Xcm

WithdrawAsset(assets: **MultiAssets**, effects: Vec<Order<Call>>)
ReserveAssetDeposit**ed**(assets: **MultiAssets**, effects: Vec<Order<Call>>)
ReceiveTeleportedAsset(assets: **MultiAssets**, effects: Vec<Order<Call>>)
QueryResponse(query_id: u64, response: Response)
TransferAsset(assets: **MultiAssets**, beneficiary: MultiLocation)
TransferReserveAsset(assets: **MultiAssets**, dest: MultiLocation, effects: Vec<Order<()>>)
Transact(origin_type: OriginKind, require_weight_at_most: u64, call: DoubleEncoded<Call>)
HrmpNewChannelOpenRequest(sender: u32, max_message_size: u32, max_capacity: u32)
HrmpChannelAccepted(recipient: u32)
HrmpChannelClosing(initiator: u32, sender: u32, recipient: u32)
RelayedFrom(who: **InteriorMultiLocation**, message: alloc::boxed::Box<Xcm<Call>>)
SubscribeVersion(query_id: u64, max_response_weight: u64)
UnsubscribeVersion()

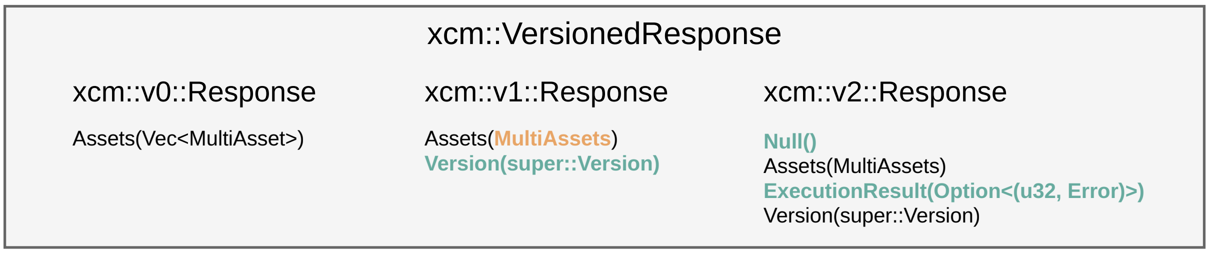
xcm::v1::order::Order

Noop()
DepositAsset(assets: **MultiAssetFilter**, **max_assets: u32,** beneficiary: MultiLocation)
DepositReserveAsset(assets: **MultiAssetFilter**, max_assets: u32, dest: MultiLocation, effects: Vec<Order<()>>)
ExchangeAsset(give: **MultiAssetFilter**, receive: **MultiAssets**)
InitiateReserveWithdraw(assets: **MultiAssetFilter**, reserve: MultiLocation, effects: Vec<Order<()>>)
InitiateTeleport(assets: **MultiAssetFilter**, dest: MultiLocation, effects: Vec<Order<()>>)
QueryHolding(query_id: u64, dest: MultiLocation, assets: **MultiAssetFilter**)
BuyExecution(fees: MultiAsset, weight: u64, debt: u64, halt_on_error: bool, instructions: Vec<Xcm<Call>>)

xcm::v2::Instruction

WithdrawAsset(MultiAssets~~, effects: Vec<Order<Call>>~~)
ReserveAssetDeposited(MultiAssets~~, effects: Vec<Order<Call>>~~)
ReceiveTeleportedAsset(MultiAssets~~, effects: Vec<Order<Call>>~~)
QueryResponse(query_id: **QueryId**, response: Response, **max_weight: u64**)
TransferAsset(assets: MultiAssets, beneficiary: MultiLocation)
TransferReserveAsset(assets: MultiAssets, dest: MultiLocation~~, effects: Vec<Order<()>>~~, **xcm: Xcm<()>**)
Transact(origin_type: OriginKind, require_weight_at_most: u64, call: DoubleEncoded<Call>)
HrmpNewChannelOpenRequest(sender: u32, max_message_size: u32, max_capacity: u32)
HrmpChannelAccepted(recipient: u32)
HrmpChannelClosing(initiator: u32, sender: u32, recipient: u32)
~~RelayedFrom(who: InteriorMultiLocation, message: alloc::boxed::Box<Xcm<Call>>)~~
SubscribeVersion(query_id: **QueryId**, max_response_weight: u64)
UnsubscribeVersion()
ClearOrigin()
ClearError()
DescendOrigin(InteriorMultiLocation)
ReportError(query_id: QueryId, dest: MultiLocation, max_response_weight: u64)
DepositAsset(assets: MultiAssetFilter, max_assets: u32, beneficiary: MultiLocation)
DepositReserveAsset(assets: MultiAssetFilter, max_assets: u32, dest: MultiLocation, ~~effects: Vec<Order<()>>,~~ **xcm: Xcm<()>**)
ExchangeAsset(give: MultiAssetFilter, receive: MultiAssets)
InitiateReserveWithdraw(assets: MultiAssetFilter, reserve: MultiLocation, ~~effects: Vec<Order<()>>~~, **xcm: Xcm<()>**)
InitiateTeleport(assets: MultiAssetFilter, dest: MultiLocation, ~~effects: Vec<Order<()>>,~~ **xcm: Xcm<()>**)
QueryHolding(query_id: **QueryId**, dest: MultiLocation, assets: MultiAssetFilter**, max_response_weight: u64**)
BuyExecution(fees: MultiAsset, **weight_limit: WeightLimit**, ~~debt: u64, halt_on_erro: bool, instructions: Vec<Xcm<Call>>~~)
RefundSurplus()
SetErrorHandler(Xcm<Call>)
SetAppendix(Xcm<Call>)
ClaimAsset(assets: MultiAssets, ticket: MultiLocation)
Trap(u64)

Figure B.3: XCM Instructions



xcm::VersionedResponse

xcm::v0::Response

Assets(Vec<MultiAsset>)

xcm::v1::Response

Assets(**MultiAssets**)
Version(super::Version)

xcm::v2::Response

Null()
Assets(MultiAssets)
ExecutionResult(Option<(u32, Error)>)
Version(super::Version)

Figure B.4: XCM Response

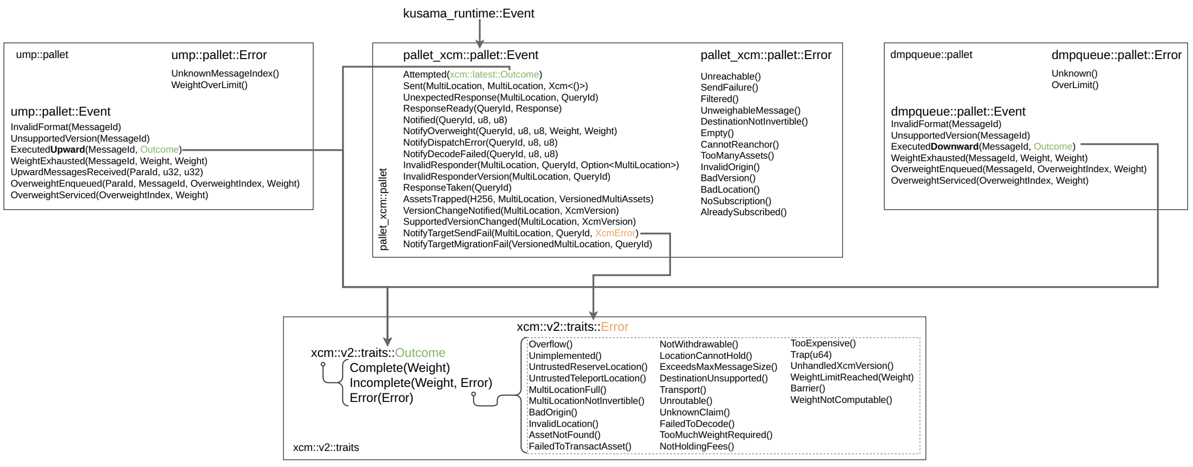

kusama_runtime::Event
ump::pallet
ump::pallet::Error
UnknownMessageIndex()
WeightOverLimit()
ump::pallet::Event
InvalidFormat(MessageId)
UnsupportedVersion(MessageId)
ExecutedUpward(MessageId, Outcome)
WeightExhausted(MessageId, Weight, Weight)
UpwardMessagesReceived(ParaId, u32, u32)
OverweightEnqueued(ParaId, MessageId, OverweightIndex, Weight)
OverweightServiced(OverweightIndex, Weight)
pallet_xcm::pallet
pallet_xcm::pallet::Event
Attempted(xcm::latest::Outcome)
Sent(MultiLocation, MultiLocation, Xcm<()>)
UnexpectedResponse(MultiLocation, QueryId)
ResponseReady(QueryId, Response)
Notified(QueryId, u8, u8)
NotifyOverweight(QueryId, u8, u8, Weight, Weight)
NotifyDispatchError(QueryId, u8, u8)
NotifyDecodeFailed(QueryId, u8, u8)
InvalidResponder(MultiLocation, QueryId, Option<MultiLocation>)
InvalidResponderVersion(MultiLocation, QueryId)
ResponseTaken(QueryId)
AssetsTrapped(H256, MultiLocation, VersionedMultiAssets)
VersionChangeNotified(MultiLocation, XcmVersion)
SupportedVersionChanged(MultiLocation, XcmVersion)
NotifyTargetSendFail(MultiLocation, QueryId, XcmError)
NotifyTargetMigrationFail(VersionedMultiLocation, QueryId)
pallet_xcm::pallet::Error
Unreachable()
SendFailure()
Filtered()
UnweighableMessage()
DestinationNotInvertible()
Empty()
CannotReanchor()
TooManyAssets()
InvalidOrigin()
BadVersion()
BadLocation()
NoSubscription()
AlreadySubscribed()
dmpqueue::pallet
dmpqueue::pallet::Error
Unknown()
OverLimit()
dmpqueue::pallet::Event
InvalidFormat(MessageId)
UnsupportedVersion(MessageId)
ExecutedDownward(MessageId, Outcome)
WeightExhausted(MessageId, Weight, Weight)
OverweightEnqueued(MessageId, OverweightIndex, Weight)
OverweightServiced(OverweightIndex, Weight)
xcm::v2::traits::Outcome
Complete(Weight)
Incomplete(Weight, Error)
Error(Error)
xcm::v2::traits::Error
Overflow()
Unimplemented()
UntrustedReserveLocation()
UntrustedTeleportLocation()
MultiLocationFull()
MultiLocationNotInvertible()
BadOrigin()
InvalidLocation()
AssetNotFound()
FailedToTransactAsset()
NotWithdrawable()
LocationCannotHold()
ExceedsMaxMessageSize()
DestinationUnsupported()
Transport()
Unroutable()
UnknownClaim()
FailedToDecode()
TooMuchWeightRequired()
NotHoldingFees()
TooExpensive()
Trap(u64)
UnhandledXcmVersion()
WeightLimitReached(Weight)
Barrier()
WeightNotComputable()
xcm::v2::traits

Figure B.5: XCM Events and common traits