libp2p-rendezvous 0.17.1

..
generated
client.rs
codec.rs
lib.rs
server.rs

```rust
// Copyright 2021 COMIT Network.
//
// Permission is hereby granted, free of charge, to any person obtaining a
// copy of this software and associated documentation files (the "Software"),
// to deal in the Software without restriction, including without limitation
// the rights to use, copy, modify, merge, publish, distribute, sublicense,
// and/or sell copies of the Software, and to permit persons to whom the
// Software is furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in
// all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
// OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
// FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
// DEALINGS IN THE SOFTWARE.

use std::{
    collections::{HashMap, VecDeque},
    iter,
    task::{Context, Poll},
    time::Duration,
};

use futures::{
    future::{BoxFuture, FutureExt},
    stream::{FuturesUnordered, StreamExt},
};
use libp2p_core::{transport::PortUse, Endpoint, Multiaddr, PeerRecord};
use libp2p_identity::{Keypair, PeerId, SigningError};
use libp2p_request_response::{OutboundRequestId, ProtocolSupport};
use libp2p_swarm::{
    ConnectionDenied, ConnectionId, ExternalAddresses, FromSwarm, NetworkBehaviour, T
```

```rust
};

use crate::codec::{
    Cookie, ErrorCode, Message, Message::*, Namespace, NewRegistration, Registration,
};

pub struct Behaviour {
    events: VecDeque<ToSwarm<<Self as NetworkBehaviour>::ToSwarm, THandlerInEvent<Sel

    inner: libp2p_request_response::Behaviour<crate::codec::Codec>,

    keypair: Keypair,

    waiting_for_register: HashMap<OutboundRequestId, (PeerId, Namespace)>,
    waiting_for_discovery: HashMap<OutboundRequestId, (PeerId, Option<Namespace>)>,

    /// Hold addresses of all peers that we have discovered so far.
    ///
    /// Storing these internally allows us to assist the [`libp2p_swarm::Swarm`] in d
    /// returning addresses from [`NetworkBehaviour::handle_pending_outbound_connecti
    discovered_peers: HashMap<PeerId, HashMap<Namespace, Vec<Multiaddr>>>,

    registered_namespaces: HashMap<(PeerId, Namespace), Ttl>,

    /// Tracks the expiry of registrations that we have discovered and stored in `dis
    /// otherwise we have a memory leak.
    expiring_registrations: FuturesUnordered<BoxFuture<'static, (PeerId, Namespace)>>

    external_addresses: ExternalAddresses,
}

impl Behaviour {
    /// Create a new instance of the rendezvous [`NetworkBehaviour`].
    pub fn new(keypair: Keypair) -> Self {
        Self {
            events: Default::default(),
            inner: libp2p_request_response::Behaviour::with_codec(
                crate::codec::Codec::default(),
                iter::once((crate::PROTOCOL_IDENT, ProtocolSupport::Outbound)),
                libp2p_request_response::Config::default(),
            ),
            keypair,
            waiting_for_register: Default::default(),
            waiting_for_discovery: Default::default(),
            discovered_peers: Default::default(),
            registered_namespaces: Default::default(),
            expiring_registrations: FuturesUnordered::from_iter(vec![
                futures::future::pending().boxed()
            ]),
            external_addresses: Default::default(),
        }
```

```rust
    /// Register our external addresses in the given namespace with the given rendezv
    ///
    /// External addresses are either manually added via
    /// [`libp2p_swarm::Swarm::add_external_address`] or reported by other [`NetworkB
    /// via [`ToSwarm::ExternalAddrConfirmed`].
    pub fn register(
        &mut self,
        namespace: Namespace,
        rendezvous_node: PeerId,
        ttl: Option<Ttl>,
    ) -> Result<(), RegisterError> {
        let external_addresses = self.external_addresses.iter().cloned().collect::<Ve
        if external_addresses.is_empty() {
            return Err(RegisterError::NoExternalAddresses);
        }

        let peer_record = PeerRecord::new(&self.keypair, external_addresses)?;
        let req_id = self.inner.send_request(
            &rendezvous_node,
            Register(NewRegistration::new(namespace.clone(), peer_record, ttl)),
        );
        self.waiting_for_register
            .insert(req_id, (rendezvous_node, namespace));

        Ok(())
    }

    /// Unregister ourselves from the given namespace with the given rendezvous peer.
    pub fn unregister(&mut self, namespace: Namespace, rendezvous_node: PeerId) {
        self.registered_namespaces
            .retain(|(rz_node, ns), _| rz_node.ne(&rendezvous_node) && ns.ne(&namespa

        self.inner
            .send_request(&rendezvous_node, Unregister(namespace));
    }

    /// Discover other peers at a given rendezvous peer.
    ///
    /// If desired, the registrations can be filtered by a namespace.
    /// If no namespace is given, peers from all namespaces will be returned.
    /// A successfully discovery returns a cookie within [`Event::Discovered`].
    /// Such a cookie can be used to only fetch the _delta_ of registrations since
    /// the cookie was acquired.
    pub fn discover(
        &mut self,
        namespace: Option<Namespace>,
        cookie: Option<Cookie>,
        limit: Option<u64>,
        rendezvous_node: PeerId,
    ) {
```

```
143             Discover {
144                 namespace: namespace.clone(),
145                 cookie,
146                 limit,
147             },
148         );
149 
150         self.waiting_for_discovery
151             .insert(req_id, (rendezvous_node, namespace));
152     }
153 }
154 
155 #[derive(Debug, thiserror::Error)]
156 pub enum RegisterError {
157     #[error("We don't know about any externally reachable addresses of ours")]
158     NoExternalAddresses,
159     #[error("Failed to make a new PeerRecord")]
160     FailedToMakeRecord(#[from] SigningError),
161 }
162 
163 #[derive(Debug)]
164 #[allow(clippy::large_enum_variant)]
165 pub enum Event {
166     /// We successfully discovered other nodes with using the contained rendezvous no
167     Discovered {
168         rendezvous_node: PeerId,
169         registrations: Vec<Registration>,
170         cookie: Cookie,
171     },
172     /// We failed to discover other nodes on the contained rendezvous node.
173     DiscoverFailed {
174         rendezvous_node: PeerId,
175         namespace: Option<Namespace>,
176         error: ErrorCode,
177     },
178     /// We successfully registered with the contained rendezvous node.
179     Registered {
180         rendezvous_node: PeerId,
181         ttl: Ttl,
182         namespace: Namespace,
183     },
184     /// We failed to register with the contained rendezvous node.
185     RegisterFailed {
186         rendezvous_node: PeerId,
187         namespace: Namespace,
188         error: ErrorCode,
189     },
190     /// The connection details we learned from this node expired.
191     Expired { peer: PeerId },
192 }
```

docs.rs ▼ Rust ▼

```
impl NetworkBehaviour for Behaviour {
    type ConnectionHandler = <libp2p_request_response::Behaviour<
        crate::codec::Codec,
    > as NetworkBehaviour>::ConnectionHandler;

    type ToSwarm = Event;

    fn handle_established_inbound_connection(
        &mut self,
        connection_id: ConnectionId,
        peer: PeerId,
        local_addr: &Multiaddr,
        remote_addr: &Multiaddr,
    ) -> Result<THandler<Self>, ConnectionDenied> {
        self.inner.handle_established_inbound_connection(
            connection_id,
            peer,
            local_addr,
            remote_addr,
        )
    }

    fn handle_established_outbound_connection(
        &mut self,
        connection_id: ConnectionId,
        peer: PeerId,
        addr: &Multiaddr,
        role_override: Endpoint,
        port_use: PortUse,
    ) -> Result<THandler<Self>, ConnectionDenied> {
        self.inner.handle_established_outbound_connection(
            connection_id,
            peer,
            addr,
            role_override,
            port_use,
        )
    }

    fn on_connection_handler_event(
        &mut self,
        peer_id: PeerId,
        connection_id: ConnectionId,
        event: THandlerOutEvent<Self>,
    ) {
        self.inner
            .on_connection_handler_event(peer_id, connection_id, event);
    }

    fn on_swarm_event(&mut self, event: FromSwarm) {
        let changed = self.external_addresses.on_swarm_event(&event);
```

```
            self.inner.on_swarm_event(event);

            if changed && self.external_addresses.iter().count() > 0 {
                let registered = self.registered_namespaces.clone();
                for ((rz_node, ns), ttl) in registered {
                    if let Err(e) = self.register(ns, rz_node, Some(ttl)) {
                        tracing::warn!("refreshing registration failed: {e}")
                    }
                }
            }
        }

        #[tracing::instrument(level = "trace", name = "NetworkBehaviour::poll", skip(self
        fn poll(
            &mut self,
            cx: &mut Context<'_>,
        ) -> Poll<ToSwarm<Self::ToSwarm, THandlerInEvent<Self>>> {
            use libp2p_request_response as req_res;
            loop {
                if let Some(event) = self.events.pop_front() {
                    return Poll::Ready(event);
                }

                match self.inner.poll(cx) {
                    Poll::Ready(ToSwarm::GenerateEvent(req_res::Event::Message {
                        message:
                            req_res::Message::Response {
                                request_id,
                                response,
                            },
                        ..
                    })) => {
                        if let Some(event) = self.handle_response(&request_id, response)
                            return Poll::Ready(ToSwarm::GenerateEvent(event));
                        }

                        continue; // not a request we care about
                    }
                    Poll::Ready(ToSwarm::GenerateEvent(req_res::Event::OutboundFailure {
                        request_id,
                        ..
                    })) => {
                        if let Some(event) = self.event_for_outbound_failure(&request_id)
                            return Poll::Ready(ToSwarm::GenerateEvent(event));
                        }

                        continue; // not a request we care about
                    }
                    Poll::Ready(ToSwarm::GenerateEvent(
                        req_res::Event::InboundFailure { .. }
                        | req_res::Event::ResponseSent { .. }
```

```
                        ..
                    },
                )) => {
                    unreachable!("rendezvous clients never receive requests")
                }
                Poll::Ready(other) => {
                    let new_to_swarm =
                        other.map_out(|_| unreachable!("we manually map `GenerateEven

                    return Poll::Ready(new_to_swarm);
                }
                Poll::Pending => {}
            }

            if let Poll::Ready(Some((peer, expired_registration))) =
                self.expiring_registrations.poll_next_unpin(cx)
            {
                let Some(registrations) = self.discovered_peers.get_mut(&peer) else {
                    continue;
                };
                registrations.remove(&expired_registration);
                if registrations.is_empty() {
                    self.discovered_peers.remove(&peer);
                }
                return Poll::Ready(ToSwarm::GenerateEvent(Event::Expired { peer }));
            }

            return Poll::Pending;
        }
    }

    fn handle_pending_outbound_connection(
        &mut self,
        _connection_id: ConnectionId,
        maybe_peer: Option<PeerId>,
        _addresses: &[Multiaddr],
        _effective_role: Endpoint,
    ) -> Result<Vec<Multiaddr>, ConnectionDenied> {
        let addrs = maybe_peer
            .map(|peer| self.discovered_peer_addrs(&peer).cloned().collect())
            .unwrap_or_default();
        Ok(addrs)
    }
}

impl Behaviour {
    fn event_for_outbound_failure(&mut self, req_id: &OutboundRequestId) -> Option<Ev
        if let Some((rendezvous_node, namespace)) = self.waiting_for_register.remove(
            return Some(Event::RegisterFailed {
                rendezvous_node,
```

```
                error: ErrorCode::Unavailable,
            });
        };

        if let Some((rendezvous_node, namespace)) = self.waiting_for_discovery.remove
            return Some(Event::DiscoverFailed {
                rendezvous_node,
                namespace,
                error: ErrorCode::Unavailable,
            });
        };

        None
    }

    fn handle_response(
        &mut self,
        request_id: &OutboundRequestId,
        response: Message,
    ) -> Option<Event> {
        match response {
            RegisterResponse(Ok(ttl)) => {
                let (rendezvous_node, namespace) = self.waiting_for_register.remove(r
                self.registered_namespaces
                    .insert((rendezvous_node, namespace.clone()), ttl);

                Some(Event::Registered {
                    rendezvous_node,
                    ttl,
                    namespace,
                })
            }
            RegisterResponse(Err(error_code)) => {
                let (rendezvous_node, namespace) = self.waiting_for_register.remove(r
                Some(Event::RegisterFailed {
                    rendezvous_node,
                    namespace,
                    error: error_code,
                })
            }
            DiscoverResponse(Ok((registrations, cookie))) => {
                let (rendezvous_node, _ns) = self.waiting_for_discovery.remove(reques
                registrations.iter().for_each(|registration| {
                    let peer_id = registration.record.peer_id();
                    let addresses = registration.record.addresses();
                    let namespace = registration.namespace.clone();
                    let ttl = registration.ttl;

                    // Emit events for all newly discovered addresses.
                    let new_addr_events = addresses
                        .iter()
```

```
                                return None;
                            }
                            Some(ToSwarm::NewExternalAddrOfPeer {
                                peer_id,
                                address: address.clone(),
                            })
                        })
                        .collect::<Vec<_>>();
                    self.events.extend(new_addr_events);

                    // Update list of discovered peers.
                    self.discovered_peers
                        .entry(peer_id)
                        .or_default()
                        .insert(namespace.clone(), addresses.to_owned());

                    // Push registration expiry future.
                    self.expiring_registrations.push(
                        async move {
                            // if the timer errors we consider it expired
                            futures_timer::Delay::new(Duration::from_secs(ttl)).await
                            (peer_id, namespace)
                        }
                        .boxed(),
                    );
                });

                Some(Event::Discovered {
                    rendezvous_node,
                    registrations,
                    cookie,
                })
            }
            DiscoverResponse(Err(error_code)) => {
                let (rendezvous_node, ns) = self.waiting_for_discovery.remove(request
                Some(Event::DiscoverFailed {
                    rendezvous_node,
                    namespace: ns,
                    error: error_code,
                })
            }
            _ => unreachable!("rendezvous clients never receive requests"),
        }
    }

    fn discovered_peer_addrs(&self, peer: &PeerId) -> impl Iterator<Item = &Multiaddr
        self.discovered_peers
            .get(peer)
            .map(|addrs| addrs.values().flatten())
```