BLOCKCHAIN COMMONS

GSTP:
GORDIAN SEALED TRANSACTION PROTOCOL

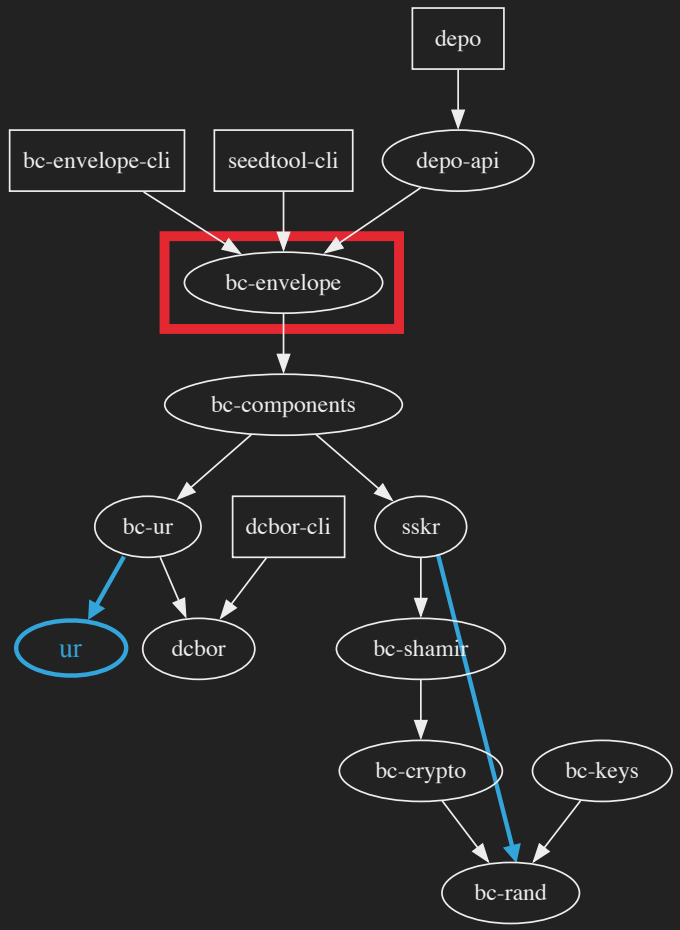
THANK YOU TO OUR RESEARCH SPONSOR

THANK YOU TO OUR RESEARCH SPONSOR

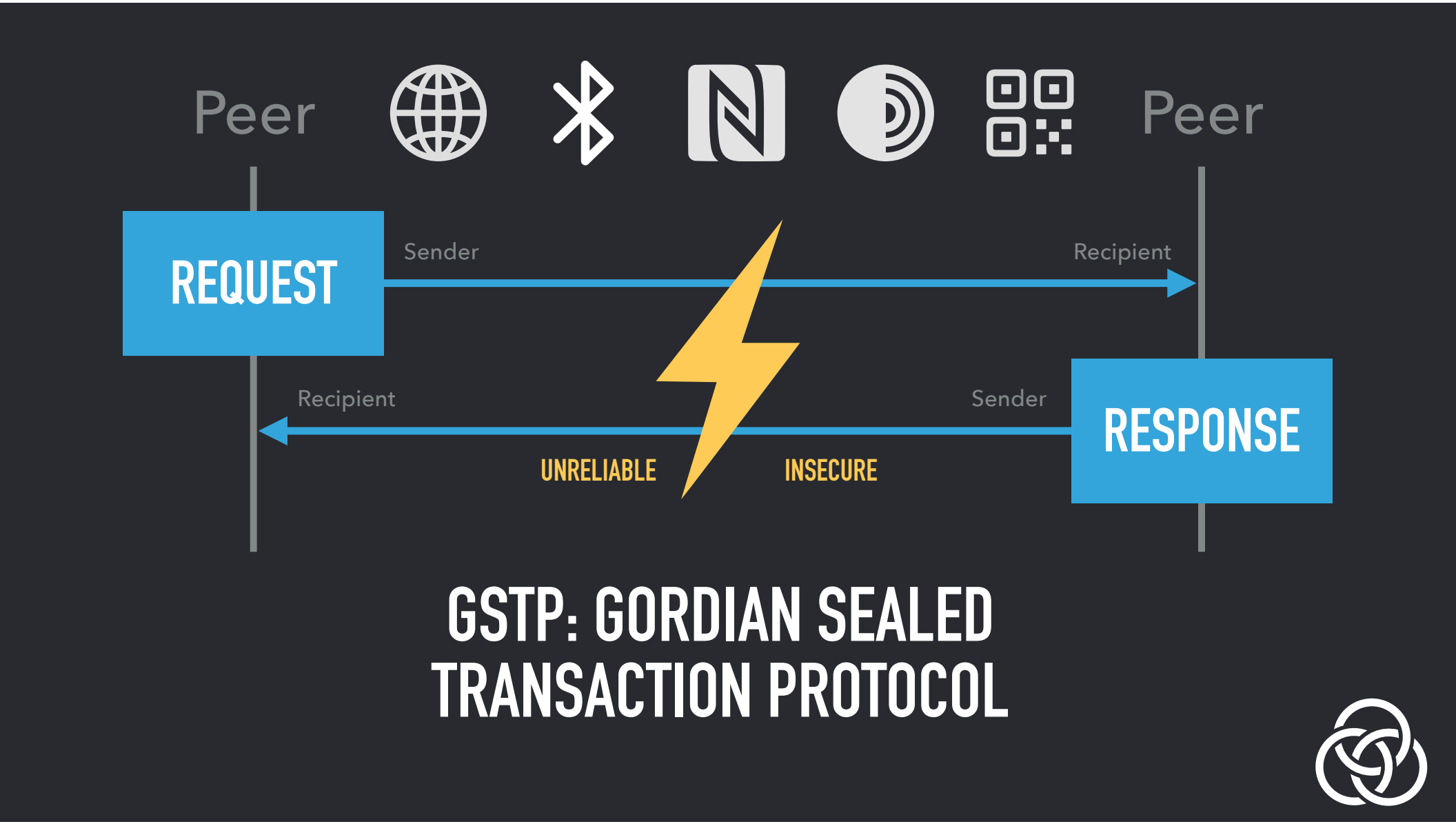
FOUNDATION

BC RUST STACK

- Executables are rectangles, libraries are ovals.
- Transitive dependencies are omitted.
  - For example, edges from other libraries that depend directly on bc-rand are omitted, since everything depends on it at least indirectly.
- The bc-ur crate is based on a Dominik Spicher's ur crate.
  - Another way the open source community supports our work.
  - Our version adds direct support for dCBOR.
- Our support for GSTP is primarily located in the bc-envelope crate.
  - Gated behind a Rust feature, on by default.



GSTP OVERVIEW



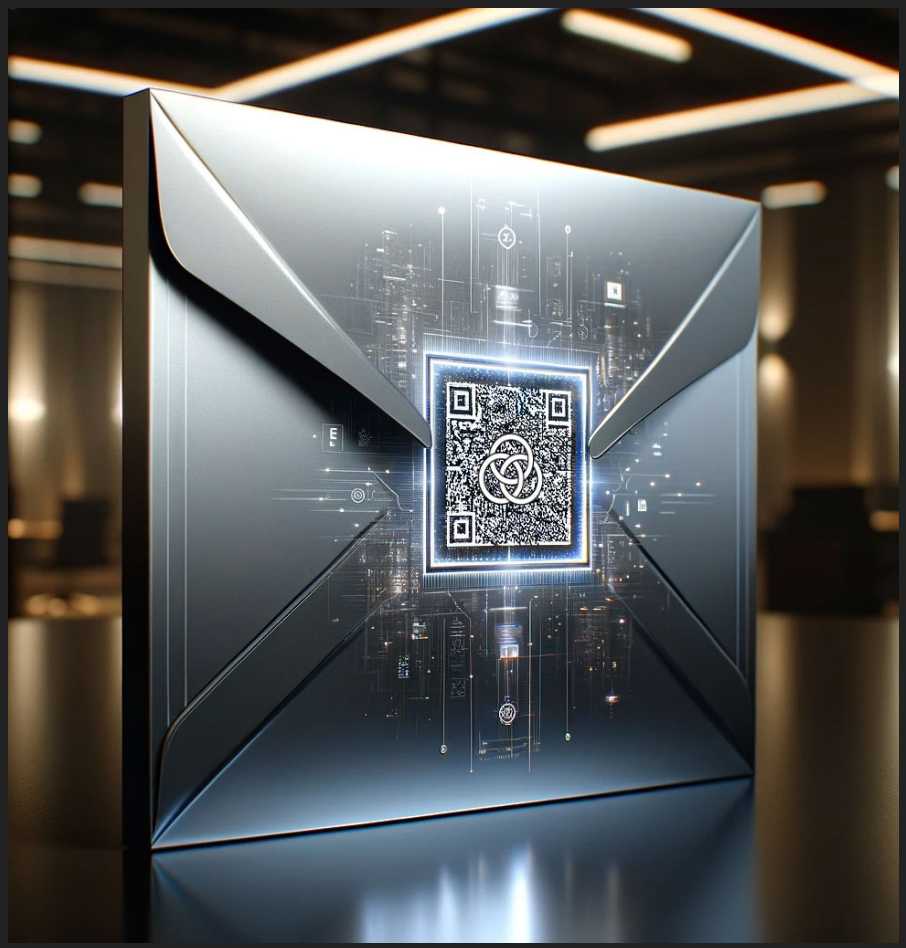
GSTP: GORDIAN SEALED TRANSACTION PROTOCOL

GOALS AND KEY COMPONENTS

- **Support asynchronous communication** between peers as well as client-server
- **Support multimodal transactions** (using more than one channel as desired)
- **Unique IDs** for request-response pairs
- **Bidirectional encryption** using public-key cryptography, including encrypted multicast
- **Self-signed requests** with included public keys for authentication
- **Message-encapsulated state** via self-encrypted "continuations"
- **Expected response ID** and **continuation timeout** mechanisms included in continuations to resist replays
- Encourage **idempotent actions** for consistency and reliability

BENEFITS

- **Reduced State Management:** Continuations allow peers to offload private state information into the passed messages, minimizing (ideally eliminating) the amount of local state needed for ongoing tasks.
- **Fault Tolerance:** Idempotency of actions ensures that the system remains consistent even if actions are repeated due to lost messages.
- **Scalability:** Responders can process requests independently and return continuations, enabling horizontal scalability and load distribution.
- **Security:** Self-encrypted continuations maintain the privacy and integrity of the sender's state information. The inclusion of expected response IDs and timeouts in continuations helps prevent misuse or malicious manipulation of continuations, e.g., replay attacks.
- **Efficient Data Handling:** Continuations in responses enable pagination, streaming, and incremental processing of large result sets, optimizing resource utilization and performance.



REQUESTS AND RESPONSES AS DISTRIBUTED FUNCTION CALLS

```
async fn do_it(arg1: Type1, arg2: Type2, …) -> Result<Success, Failure>
```

```
async fn do_it(arg1: Type1, arg2: Type2, …) -> Result<Success, Failure>
```

```
«do_it» [
    ⦇arg1⦈: Type1
    ⦇arg2⦈: Type2
    …
]
```

ENVELOPE EXPRESSION

```
async fn do_it(arg1: Type1, arg2: Type2, …) -> Result<Success, Failure>
```

REQUEST

```
request(ARID(c66be27d)) [
    'body': «do_it» [
        ❰arg1❱: Type1
        ❰arg2❱: Type2
        …
    ]
]
```

```
async fn do_it(arg1: Type1, arg2: Type2, …) -> Result<Success, Failure>
```

REQUEST

```
request(ARID(c66be27d)) [
    'body': «do_it» [
        ❰arg1❱: Type1
        ❰arg2❱: Type2
        …
    ]
]
```

RESPONSE

```
response(ARID(c66be27d)) [
    'result': Success
]

response(ARID(c66be27d)) [
    'error': Failure
]
```

```
async fn do_it(arg1: Type1, arg2: Type2, …) -> Result<Success, Failure>
```

REQUEST

```
request(ARID(c66be27d)) [
    'body': «do_it» [
        ❰arg1❱: Type1
        ❰arg2❱: Type2
        ...
    ]
]
```

RESPONSE

```
response(ARID(c66be27d)) [
    'result': Success
]

response(ARID(c66be27d)) [
    'error': Failure
]
```

async fn do_it(arg1: Type1, arg2: Type2, …) -> Result<Success, Failure>

REQUEST

```
request(ARID(c66be27d)) [
    'body': «do_it» [
        ❰arg1❱: Type1
        ❰arg2❱: Type2
        …
    ]
]
```

RESPONSE

```
response(ARID(c66be27d)) [
    'result': Success
]

response(ARID(c66be27d)) [
    'error': Failure
]
```

```
async fn do_it(arg1: Type1, arg2: Type2, …) -> Result<Success, Failure>
```

REQUEST

```
request(ARID(c66be27d)) [
    'body': «do_it» [
        ❰arg1❱: Type1
        ❰arg2❱: Type2
        ...
    ]
]
```

RESPONSE

```
response(ARID(c66be27d)) [
    'result': Success
]

response(ARID(c66be27d)) [
    'error': Failure
]
```

async fn do_it(arg1: Type1, arg2: Type2, …) -> Result<Success, Failure>

REQUEST

```
request(ARID(c66be27d)) [
    'body': «do_it» [
        ❰arg1❱: Type1
        ❰arg2❱: Type2
        ...
    ]
]
```

RESPONSE

```
response(ARID(c66be27d)) [
    'result': Success
]

response(ARID(c66be27d)) [
    'error': Failure
]
```

async fn do_it(arg1: Type1, arg2: Type2, …) -> Result<Success, Failure>

REQUEST

```
request(ARID(c66be27d)) [
    'body': «do_it» [
        ❰arg1❱: Type1
        ❰arg2❱: Type2
        …
    ]
]
```

EXPRESSION

RESPONSE

```
response(ARID(c66be27d)) [
    'result': Success
]

response(ARID(c66be27d)) [
    'error': Failure
]
```

```
async fn do_it(arg1: Type1, arg2: Type2, …) -> Result<Success, Failure>
```

REQUEST

```
request(ARID(c66be27d)) [
    'body': «do_it» [
        ❰arg1❱: Type1
        ❰arg2❱: Type2
        ...
    ]
]
```

RESPONSE

```
response(ARID(c66be27d)) [
    'result': Success
]

response(ARID(c66be27d)) [
    'error': Failure
]
```

SENDER →

REQUEST

```
request(ARID(c66be27d)) [
    'body': «do_it» [
        ❪arg1❫: Type1
        ❪arg2❫: Type2
        ...
    ]
]
```

SENDER →

REQUEST

AUTHENTICATION

```
request(ARID(c66be27d)) [
    'body': «do_it» [
        ❪arg1❫: Type1
        ❪arg2❫: Type2
        ...
    ]
]
```

SENDER →

REQUEST

AUTHENTICATION

```
request(ARID(c66be27d)) [
    'body': «do_it» [
        ❪arg1❫: Type1
        ❪arg2❫: Type2
        ...
    ]
    'senderPublicKey': PublicKeyBase
]
```

SENDER →

REQUEST

AUTHENTICATION

```
request(ARID(c66be27d)) [
    'body': «do_it» [
        ❪arg1❫: Type1
        ❪arg2❫: Type2
        ...
    ]
    'senderPublicKey': PublicKeyBase
    'senderContinuation': State
]
```

SENDER →

REQUEST

AUTHENTICATION

```
request(ARID(c66be27d)) [
    'body': «do_it» [
        ❪arg1❫: Type1
        ❪arg2❫: Type2
        ...
    ]
    'senderPublicKey': PublicKeyBase
    'senderContinuation': ENCRYPTED
]
```

SENDER →

REQUEST

AUTHENTICATION

```
request(ARID(c66be27d)) [
    'body': «do_it» [
        ❪arg1❫: Type1
        ❪arg2❫: Type2
        ...
    ]
    'senderPublicKey': PublicKeyBase
    'senderContinuation': ENCRYPTED
    'recipientContinuation': ENCRYPTED
]
```

SENDER →

REQUEST

AUTHENTICATION

```
request(ARID(c66be27d)) [
    'body': «do_it» [
        ❪arg1❫: Type1
        ❪arg2❫: Type2
        ...
    ]
    'senderPublicKey': PublicKeyBase
    'senderContinuation': ENCRYPTED
    'recipientContinuation': ENCRYPTED
    'verifiedBy': Signature
]
```

SENDER →

REQUEST

AUTHENTICATION

```
request(ARID(c66be27d)) [
    'body': «do_it» [
        ❪arg1❫: Type1
        ❪arg2❫: Type2
        ...
    ]
    'senderPublicKey': PublicKeyBase
    'senderContinuation': ENCRYPTED
    'recipientContinuation': ENCRYPTED
    'verifiedBy': Signature
]
```

SENDER →

REQUEST

AUTHENTICATION

```
request(ARID(c66be27d)) [
    'body': «do_it» [
        ❪arg1❫: Type1
        ❪arg2❫: Type2
        ...
    ]
    'senderPublicKey': PublicKeyBase
    'senderContinuation': ENCRYPTED
    'recipientContinuation': ENCRYPTED
]
```

SENDER →

REQUEST

AUTHENTICATION

```
request(ARID(c66be27d)) [
    'body': «do_it» [
        ❪arg1❫: Type1
        ❪arg2❫: Type2
        ...
    ]
    'senderPublicKey': PublicKeyBase
    'senderContinuation': ENCRYPTED
    'recipientContinuation': ENCRYPTED
]
```

SENDER →

REQUEST

AUTHENTICATION

```
{
    request(ARID(c66be27d)) [
        'body': «do_it» [
            ❰arg1❱: Type1
            ❰arg2❱: Type2
            ...
        ]
        'senderPublicKey': PublicKeyBase
        'senderContinuation': ENCRYPTED
        'recipientContinuation': ENCRYPTED
    ]
}
```

SENDER →

REQUEST

AUTHENTICATION

```
{
    request(ARID(c66be27d)) [
        'body': «do_it» [
            ❪arg1❫: Type1
            ❪arg2❫: Type2
            ...
        ]
        'senderPublicKey': PublicKeyBase
        'senderContinuation': ENCRYPTED
        'recipientContinuation': ENCRYPTED
    ]
}
```

SENDER →

REQUEST

AUTHENTICATION

```
{
    request(ARID(c66be27d)) [
        'body': «do_it» [
            ❪arg1❫: Type1
            ❪arg2❫: Type2
            ...
        ]
        'senderPublicKey': PublicKeyBase
        'senderContinuation': ENCRYPTED
        'recipientContinuation': ENCRYPTED
    ]
} [
    'verifiedBy': Signature
]
```

SENDER →


```
{
    request(ARID(c66be27d)) [
        'body': «do_it» [
            ❰arg1❱: Type1
            ❰arg2❱: Type2
            ...
        ]
        'senderPublicKey': PublicKeyBase
        'senderContinuation': ENCRYPTED
        'recipientContinuation': ENCRYPTED
    ]
} [
    'verifiedBy': Signature
]
```

SENDER →


```
{
    request(ARID(c66be27d)) [
        'body': «do_it» [
            ❪arg1❫: Type1
            ❪arg2❫: Type2
            ...
        ]
        'senderPublicKey': PublicKeyBase
        'senderContinuation': ENCRYPTED
        'recipientContinuation': ENCRYPTED
    ]
} [
    'verifiedBy': Signature
]
```

SENDER →

ENCRYPTED [
    'verifiedBy': Signature
]

SENDER →


```
{
    request(ARID(c66be27d)) [
        'body': «do_it» [
            ❪arg1❫: Type1
            ❪arg2❫: Type2
            ...
        ]
        'senderPublicKey': PublicKeyBase
        'senderContinuation': ENCRYPTED
        'recipientContinuation': ENCRYPTED
    ]
} [
    'verifiedBy': Signature
]
```

SENDER →


```
{
    request(ARID(c66be27d)) [
        'body': «do_it» [
            ❰arg1❱: Type1
            ❰arg2❱: Type2
            ...
        ]
        'senderPublicKey': PublicKeyBase
        'senderContinuation': ENCRYPTED
        'recipientContinuation': ENCRYPTED
    ]
} [
    'verifiedBy': Signature
]
```

SENDER →


```
{
    {
        request(ARID(c66be27d)) [
            'body': «do_it» [
                ❰arg1❱: Type1
                ❰arg2❱: Type2
                ...
            ]
            'senderPublicKey': PublicKeyBase
            'senderContinuation': ENCRYPTED
            'recipientContinuation': ENCRYPTED
        ]
    } [
        'verifiedBy': Signature
    ]
}
```

SENDER →


```
{
    {
        request(ARID(c66be27d)) [
            'body': «do_it» [
                ❮arg1❯: Type1
                ❮arg2❯: Type2
                ...
            ]
            'senderPublicKey': PublicKeyBase
            'senderContinuation': ENCRYPTED
            'recipientContinuation': ENCRYPTED
        ]
    } [
        'verifiedBy': Signature
    ]
}
```

SENDER →

ENCRYPTED

SENDER →


```
ENCRYPTED [
    'hasRecipient': SealedMessage
]
```

→ RECIPIENT


```
ENCRYPTED [
    'hasRecipient': SealedMessage
]
```

→RECIPIENT

REQUEST

AUTHENTICATION

```
{
    request(ARID(c66be27d)) [
        'body': «do_it» [
            ❪arg1❫: Type1
            ❪arg2❫: Type2
            ...
        ]
        'senderPublicKey': PublicKeyBase
        'senderContinuation': ENCRYPTED
        'recipientContinuation': ENCRYPTED
    ]
} [
    'verifiedBy': Signature
]
```

→ RECIPIENT

REQUEST

```
request(ARID(c66be27d)) [
    'body': «do_it» [
        ❪arg1❫: Type1
        ❪arg2❫: Type2
        ...
    ]
    'senderPublicKey': PublicKeyBase
    'senderContinuation': ENCRYPTED
    'recipientContinuation': ENCRYPTED
]
```

REQUEST

→ RECIPIENT

```
async fn do_it(arg1: Type1, arg2: Type2, ...) -> ...
```

id: ARID(c66be27d)

expression:
«do_it» [
❰arg1❱: Type1
❰arg2❱: Type2
...
]

sender: PublicKeyBase

peer_continuation: ENCRYPTED

state: ENCRYPTED

REQUEST

→ RECIPIENT

async fn do_it(arg1: Type1, arg2: Type2, ...) -> ...

id: ARID(c66be27d)

expression:
«do_it» [
❪arg1❫: Type1
❪arg2❫: Type2
...
]

sender: PublicKeyBase

peer_continuation: ENCRYPTED

state: State

←SENDER

REQUEST

id: ARID(c66be27d)

sender: PublicKeyBase

peer_continuation: ENCRYPTED

state: State

Result<Success, Failure>

RESPONSE

REQUEST

←SENDER

sender: PublicKeyBase
peer_continuation: ENCRYPTED
state: State

RESPONSE

```
response(ARID(c66be27d)) [
    'result': Success
]
```

← SENDER

REQUEST

sender: PublicKeyBase
peer_continuation: ENCRYPTED
state: State

RESPONSE

AUTHENTICATION

```
response(ARID(c66be27d)) [
    'result': Success
    'senderPublicKey': PublicKeyBase
]
```

REQUEST

←SENDER

sender: PublicKeyBase
peer_continuation: ENCRYPTED

```
response(ARID(c66be27d)) [
    'result': Success
    'senderPublicKey': PublicKeyBase
    'senderContinuation': State
]
```

RESPONSE

AUTHENTICATION

← SENDER

REQUEST

sender: PublicKeyBase
peer_continuation: ENCRYPTED

```
response(ARID(c66be27d)) [
    'result': Success
    'senderPublicKey': PublicKeyBase
    'senderContinuation': ENCRYPTED
]
```

RESPONSE

AUTHENTICATION

REQUEST

← SENDER

sender: PublicKeyBase

```
response(ARID(c66be27d)) [
    'result': Success
    'senderPublicKey': PublicKeyBase
    'senderContinuation': ENCRYPTED
    'receipientContinuation': ENCRYPTED
]
```

RESPONSE

AUTHENTICATION

← SENDER

REQUEST

sender: PublicKeyBase

```
{
    response(ARID(c66be27d)) [
        'result': Success
        'senderPublicKey': PublicKeyBase
        'senderContinuation': ENCRYPTED
        'receipientContinuation': ENCRYPTED
    ]
} [
    'verifiedBy': Signature
]
```

RESPONSE

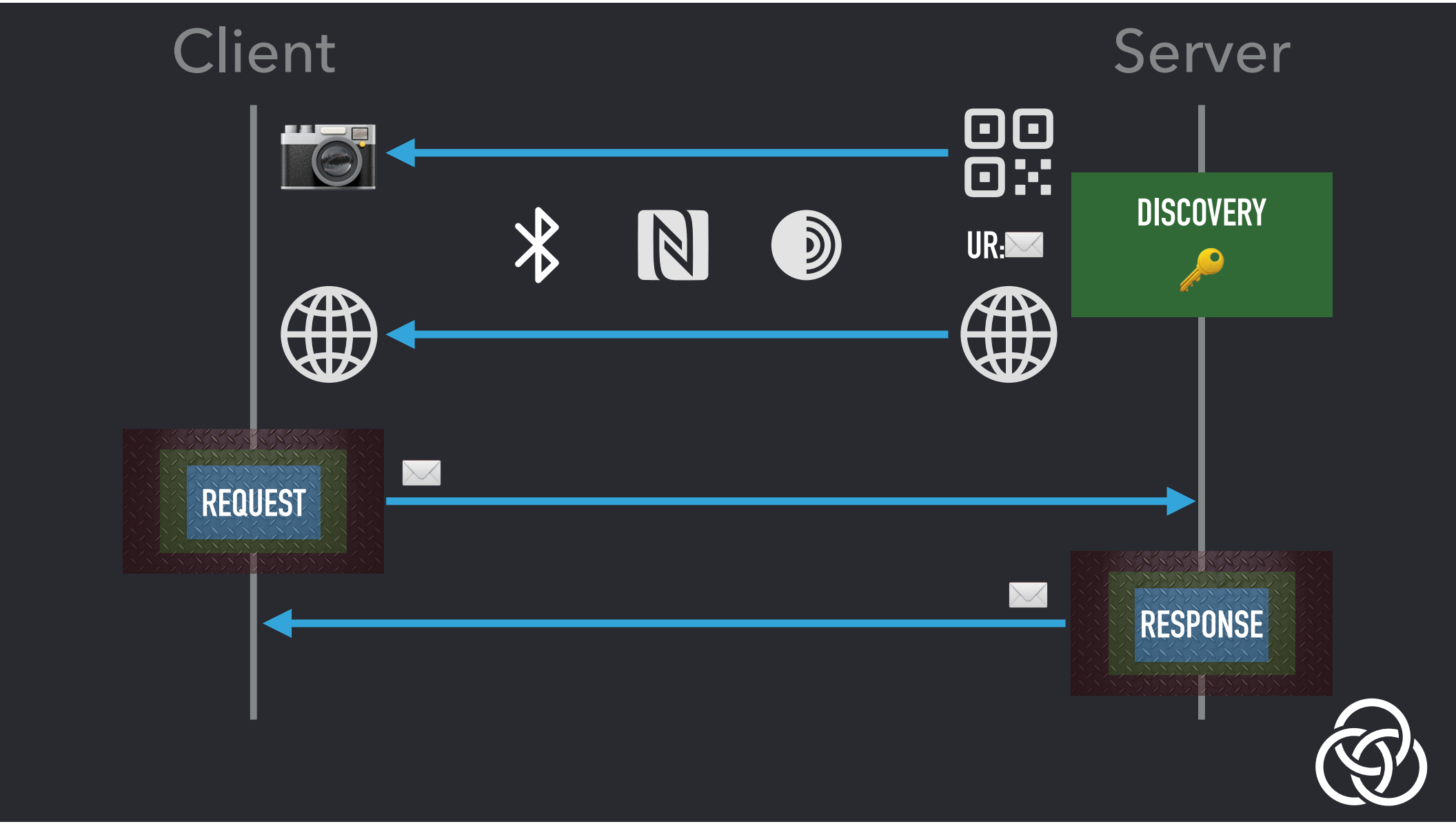
AUTHENTICATION

← SENDER

REQUEST

```
ENCRYPTED [
    'hasRecipient': SealedMessage
]
```

TRUST-ON-FIRST-USE (TOFU) PAIRING


Client
Server
UR:
DISCOVERY
REQUEST
RESPONSE

CODE WALKTHROUGH

```rust
pub async fn handle_request_string(&self, ur_string: String) -> String {
    match Envelope::from_ur_string(&ur_string) {
        Ok(request_envelope) => self.handle_request(request_envelope).await.ur_string(),
        Err(_) => {
            let sealed_response = SealedResponse::new_early_failure(self.public_key())
                .with_error("invalid request");
            Envelope::from((sealed_response, self.private_key())).ur_string()
        }
    }
}
```


```rust
pub async fn handle_request(&self, encrypted_request: Envelope) -> Envelope {
    match self.handle_unverified_request(encrypted_request).await {
        Ok(success_response) => success_response,
        Err(error) => {
            let message = error.to_string();
            SealedResponse::new_early_failure(self.public_key())
                .with_error(message)
                .into()
        },
    }
}
```

```rust
pub async fn handle_unverified_request(&self, encrypted_request: Envelope) -> Result<Envelope> {
    let sealed_request = SealedRequest::try_from((encrypted_request, self.private_key()))?;
    let id: ARID = sealed_request.id().clone();
    let function: Function = sealed_request.function().clone();
    let sender: PublicKeyBase = sealed_request.sender().clone();
    let peer_continuation: Option<Envelope> = sealed_request.peer_continuation().cloned();
    let sealed_response: SealedResponse = match self.handle_verified_request(sealed_request).await {
        Ok((result, state)) => {
            SealedResponse::new_success(id, self.public_key())
                .with_result(result)
                .with_optional_state(state)
                .with_peer_continuation(peer_continuation.as_ref())
        },
        Err(error) => {
            let function_name = function.named_name().unwrap_or("unknown".to_string()).flanked_function();
            let message = format!("{}: {} {}", id.abbrev(), function_name, error);
            SealedResponse::new_failure(id, self.public_key())
                .with_error(message)
                .with_peer_continuation(peer_continuation.as_ref())
        }
    };

    let state_expiry_date = Date::now() + Duration::from_secs(self.0.continuation_expiry_seconds());
    let sealed_envelope = Envelope::from((sealed_response, Some(&state_expiry_date), self.private_key(), &sender));
    Ok(sealed_envelope)
}
```

```rust
pub async fn handle_unverified_request(&self, encrypted_request: Envelope) -> Result<Envelope> {
    let sealed_request = SealedRequest::try_from((encrypted_request, self.private_key()))?;
    let id: ARID = sealed_request.id().clone();
    let function: Function = sealed_request.function().clone();
    let sender: PublicKeyBase = sealed_request.sender().clone();
    let peer_continuation: Option<Envelope> = sealed_request.peer_continuation().cloned();
    let sealed_response: SealedResponse = match self.handle_verified_request(sealed_request).await {
        Ok((result, state)) => {
            SealedResponse::new_success(id, self.public_key())
                .with_result(result)
                .with_optional_state(state)
                .with_peer_continuation(peer_continuation.as_ref())
        },
        Err(error) => {
            let function_name = function.named_name().unwrap_or("unknown".to_string()).flanked_function();
            let message = format!("{}: {} {}", id.abbrev(), function_name, error);
            SealedResponse::new_failure(id, self.public_key())
                .with_error(message)
                .with_peer_continuation(peer_continuation.as_ref())
        }
    };

    let state_expiry_date = Date::now() + Duration::from_secs(self.0.continuation_expiry_seconds());
    let sealed_envelope = Envelope::from((sealed_response, Some(&state_expiry_date), self.private_key(), &sender));
    Ok(sealed_envelope)
}
```

REQUEST

```rust
pub async fn handle_unverified_request(&self, encrypted_request: Envelope) -> Result<Envelope> {
    let sealed_request = SealedRequest::try_from((encrypted_request, self.private_key()))?;
    let id: ARID = sealed_request.id().clone();
    let function: Function = sealed_request.function().clone();
    let sender: PublicKeyBase = sealed_request.sender().clone();
    let peer_continuation: Option<Envelope> = sealed_request.peer_continuation().cloned();
    let sealed_response: SealedResponse = match self.handle_verified_request(sealed_request).await {
        Ok((result, state)) => {
            SealedResponse::new_success(id, self.public_key())
                .with_result(result)
                .with_optional_state(state)
                .with_peer_continuation(peer_continuation.as_ref())
        },
        Err(error) => {
            let function_name = function.named_name().unwrap_or("unknown".to_string()).flanked_function();
            let message = format!("{}: {} {}", id.abbrev(), function_name, error);
            SealedResponse::new_failure(id, self.public_key())
                .with_error(message)
                .with_peer_continuation(peer_continuation.as_ref())
        }
    };

    let state_expiry_date = Date::now() + Duration::from_secs(self.0.continuation_expiry_seconds());
    let sealed_envelope = Envelope::from((sealed_response, Some(&state_expiry_date), self.private_key(), &sender));
    Ok(sealed_envelope)
}
```

REQUEST

```rust
pub async fn handle_unverified_request(&self, encrypted_request: Envelope) -> Result<Envelope> {
    let sealed_request = SealedRequest::try_from((encrypted_request, self.private_key()))?;
    let id: ARID = sealed_request.id().clone();
    let function: Function = sealed_request.function().clone();
    let sender: PublicKeyBase = sealed_request.sender().clone();
    let peer_continuation: Option<Envelope> = sealed_request.peer_continuation().cloned();
    let sealed_response: SealedResponse = match self.handle_verified_request(sealed_request).await {
        Ok((result, state)) => {
            SealedResponse::new_success(id, self.public_key())
                .with_result(result)
                .with_optional_state(state)
                .with_peer_continuation(peer_continuation.as_ref())
        },
        Err(error) => {
            let function_name = function.named_name().unwrap_or("unknown".to_string()).flanked_function();
            let message = format!("{}: {} {}", id.abbrev(), function_name, error);
            SealedResponse::new_failure(id, self.public_key())
                .with_error(message)
                .with_peer_continuation(peer_continuation.as_ref())
        }
    };

    let state_expiry_date = Date::now() + Duration::from_secs(self.0.continuation_expiry_seconds());
    let sealed_envelope = Envelope::from((sealed_response, Some(&state_expiry_date), self.private_key(), &sender));
    Ok(sealed_envelope)
}
```

RESPONSE

```rust
pub async fn handle_unverified_request(&self, encrypted_request: Envelope) -> Result<Envelope> {
    let sealed_request = SealedRequest::try_from((encrypted_request, self.private_key()))?;
    let id: ARID = sealed_request.id().clone();
    let function: Function = sealed_request.function().clone();
    let sender: PublicKeyBase = sealed_request.sender().clone();
    let peer_continuation: Option<Envelope> = sealed_request.peer_continuation().cloned();
    let sealed_response: SealedResponse = match self.handle_verified_request(sealed_request).await {
        Ok((result, state)) => {
            SealedResponse::new_success(id, self.public_key())
                .with_result(result)
                .with_optional_state(state)
                .with_peer_continuation(peer_continuation.as_ref())
        },
        Err(error) => {
            let function_name = function.named_name().unwrap_or("unknown".to_string()).flanked_function();
            let message = format!("{}: {} {}", id.abbrev(), function_name, error);
            SealedResponse::new_failure(id, self.public_key())
                .with_error(message)
                .with_peer_continuation(peer_continuation.as_ref())
        }
    };

    let state_expiry_date = Date::now() + Duration::from_secs(self.0.continuation_expiry_seconds());
    let sealed_envelope = Envelope::from((sealed_response, Some(&state_expiry_date), self.private_key(), &sender));
    Ok(sealed_envelope)
}
```

RESPONSE

```rust
pub async fn handle_unverified_request(&self, encrypted_request: Envelope) -> Result<Envelope> {
    let sealed_request = SealedRequest::try_from((encrypted_request, self.private_key()))?;
    let id: ARID = sealed_request.id().clone();
    let function: Function = sealed_request.function().clone();
    let sender: PublicKeyBase = sealed_request.sender().clone();
    let peer_continuation: Option<Envelope> = sealed_request.peer_continuation().cloned();
    let sealed_response: SealedResponse = match self.handle_verified_request(sealed_request).await {
        Ok((result, state)) => {
            SealedResponse::new_success(id, self.public_key())
                .with_result(result)
                .with_optional_state(state)
                .with_peer_continuation(peer_continuation.as_ref())
        },
        Err(error) => {
            let function_name = function.named_name().unwrap_or("unknown".to_string()).flanked_function();
            let message = format!("{}: {} {}", id.abbrev(), function_name, error);
            SealedResponse::new_failure(id, self.public_key())
                .with_error(message)
                .with_peer_continuation(peer_continuation.as_ref())
        }
    };

    let state_expiry_date = Date::now() + Duration::from_secs(self.0.continuation_expiry_seconds());
    let sealed_envelope = Envelope::from((sealed_response, Some(&state_expiry_date), self.private_key(), &sender));
    Ok(sealed_envelope)
}
```

```rust
async fn handle_verified_request(&self, sealed_request: SealedRequest) -> Result<(Envelope, Option<Envelope>)> {
    let function: Function = sealed_request.function();
    let id: ARID = sealed_request.id();
    let sender: &PublicKeyBase = sealed_request.sender();
    let body: Expression = sealed_request.body().clone();

    let (response, state) = if function == &STORE_SHARE_FUNCTION {
        // ...
    } else if // ... {
        // ...
    } else if function == &START_RECOVERY_FUNCTION {
        let expression = StartRecovery::try_from(body)?;
        let continuation = self.start_recovery(expression.recovery(), sender).await?;
        (Envelope::ok(), Some(continuation))
    } else if function == &FINISH_RECOVERY_FUNCTION {
        let _expression = FinishRecovery::try_from(body)?;
        if let Some(state) = sealed_request.state() {
            self.finish_recovery(state, sender).await?;
        } else {
            bail!("missing state");
        }
        (Envelope::ok(), None)
    } else {
        bail!("unknown function: {}", function.name());
    };

    Ok((response, state))
}
```

REQUEST

```rust
async fn handle_verified_request(&self, sealed_request: SealedRequest) -> Result<(Envelope, Option<Envelope>)> {
    let function: Function = sealed_request.function();
    let id: ARID = sealed_request.id();
    let sender: &PublicKeyBase = sealed_request.sender();
    let body: Expression = sealed_request.body().clone();

    let (response, state) = if function == &STORE_SHARE_FUNCTION {
        // ...
    } else if // ... {
        // ...
    } else if function == &START_RECOVERY_FUNCTION {
        let expression = StartRecovery::try_from(body)?;
        let continuation = self.start_recovery(expression.recovery(), sender).await?;
        (Envelope::ok(), Some(continuation))
    } else if function == &FINISH_RECOVERY_FUNCTION {
        let _expression = FinishRecovery::try_from(body)?;
        if let Some(state) = sealed_request.state() {
            self.finish_recovery(state, sender).await?;
        } else {
            bail!("missing state");
        }
        (Envelope::ok(), None)
    } else {
        bail!("unknown function: {}", function.name());
    };

    Ok((response, state))
}
```

RESPONSE

```rust
async fn handle_verified_request(&self, sealed_request: SealedRequest) -> Result<(Envelope, Option<Envelope>)> {
    let function: Function = sealed_request.function();
    let id: ARID = sealed_request.id();
    let sender: &PublicKeyBase = sealed_request.sender();
    let body: Expression = sealed_request.body().clone();

    let (response, state) = if function == &STORE_SHARE_FUNCTION {
        // ...
    } else if // ... {
        // ...
    } else if function == &START_RECOVERY_FUNCTION {
        let expression = StartRecovery::try_from(body)?;
        let continuation = self.start_recovery(expression.recovery(), sender).await?;
        (Envelope::ok(), Some(continuation))
    } else if function == &FINISH_RECOVERY_FUNCTION {
        let _expression = FinishRecovery::try_from(body)?;
        if let Some(state) = sealed_request.state() {
            self.finish_recovery(state, sender).await?;
        } else {
            bail!("missing state");
        }
        (Envelope::ok(), None)
    } else {
        bail!("unknown function: {}", function.name());
    };

    Ok((response, state))
}
```

RESPONSE

CHRISTOPHER ALLEN
christophera@lifewithalacrity.com
@BlockchainComns

WOLF MCNALLY
wolf@wolfmcnally.com
@WolfMcNally

List of Envelope resource links:

https://www.blockchaincommons.com/introduction/Envelope-Intro/#envelope-links