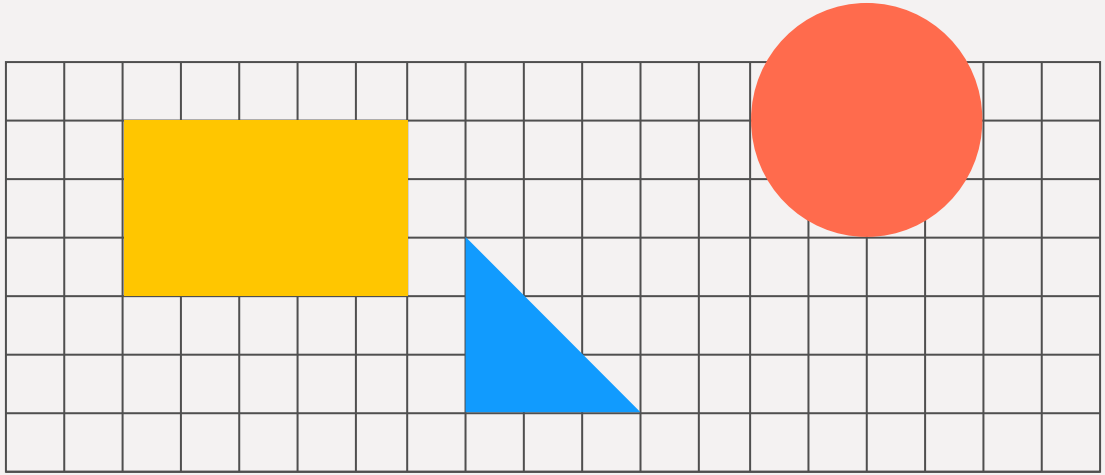
TI-Share

Demo Presentation

Group 3: Sharing Threat Intelligence using Structured P2P

Keerthana Thotakura, Maria Jothish, Sabina Sokol, Tran Ha, Seung-a Baek



Introduction

Background

Threat Intelligence

- **Definition**: The process of collecting, analyzing, and sharing information about cyber threats.
- **Types of Threats:** Includes malware, phishing attacks, security vulnerabilities, and other cyber risks.
- **Proactive Defense:** Helps organizations anticipate and mitigate threats before they cause harm.
- **Informed Decision-Making:** Provides actionable insights to improve security policies and response strategies.

Importance of Threat Intelligence Sharing

- Reputation Risk
- Competitive Concerns
- Privacy and Legal Risks
- Centralized Trust Issues

Imagine a large financial institution, **BankSecure**, detects a sophisticated phishing campaign targeting its customers. The attack exploits vulnerabilities in common online banking authentication methods. **BankSecure** wants to share this intelligence with other banks and cybersecurity organizations to prevent further damage.

Related Work

- Centralized P2P networks (e.g., early Gnutella) suffer from link congestion, single points of failure, and costly administration.
- A single peer logging and transmitting data creates vulnerabilities, including reduced efficiency and susceptibility to tampering.
- If an attacker modifies metadata collected by the only peer, the breach may go undetected due to the lack of redundancy or verification.
- A decentralized P2P system addresses these issues by:
  - Distributing data collection across multiple peers,
  - Reducing congestion,
  - Eliminating single points of failure,
  - Enabling anomaly detection through cross-peer validation.
- Iris is a decentralized P2P threat intelligence system that:
  - Shares threat data and alerts,
  - Uses DHTs and cryptographic keys for confidentiality and trust,
  - Lets peers control data privacy and dissemination.

The Prototype

Proposed Features

File Share Efficiency

- Utilize Iris for file sharing between peers on the same network
- Split files into chunks
- Peers can request a specific chunk
- A peer can reassemble chunks back into the full file

Functional Core Components

Peer-to-Peer communication (LibP2P)

- Establish direct connection between nodes
- Uses distributed hash table for decentralized peers discovery
- Manages peer directly with low water, medium water, and high water

Core protocols to exchange threat intelligence data

- Alert protocol - System alerting peers when detecting malicious activity
- Intelligence protocol - Share and validate threat intelligence
- File sharing protocol - Securely exchange malicious file

Fides Trust Model

- Determine the reliability of peers
- Peers provide feedbacks/opinions on new peers

File Sharing Algorithm

Chunked file transfer

- Files containing threat intelligence data are split into smaller chunks
- Each chunk is transferred directly between peers and reassembled on the receiving side

Metadata

- Instead of sending the file content across the network, peers first broadcast metadata (using FileShareAnnounce) to notify others of the file's availability

Distributed Hash Table (DHT)

- Peers use provide(f) to claim themselves as holders of a file f, and others use providers(f) to get a list of providers for the given key k

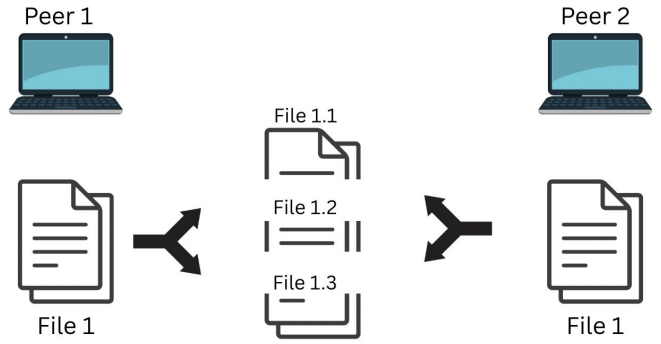
File Downloading

- Only authorized peers send FileShareDownload requests to fetch the file or specific chunks
- If the download is successful, the peers can choose if they want to be listed as another provider of the file

Feature Goals

Function 1:

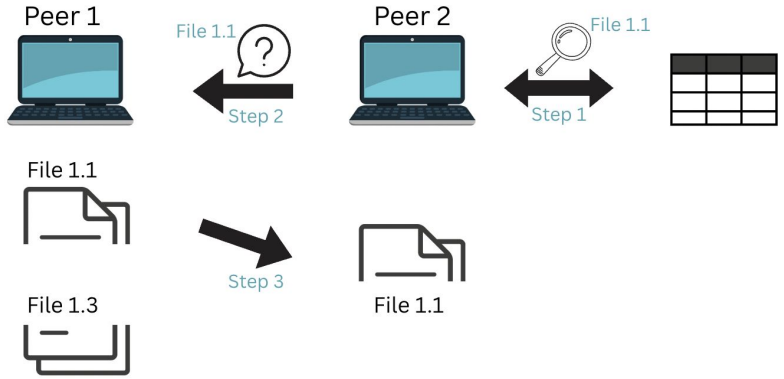
- Sending File 1 to another peer
  - Peer 1 splits file into chunks
  - Peer 1 sends chunks
  - Peer 2 receives chunks
  - Peer 2 recombines chunks to file



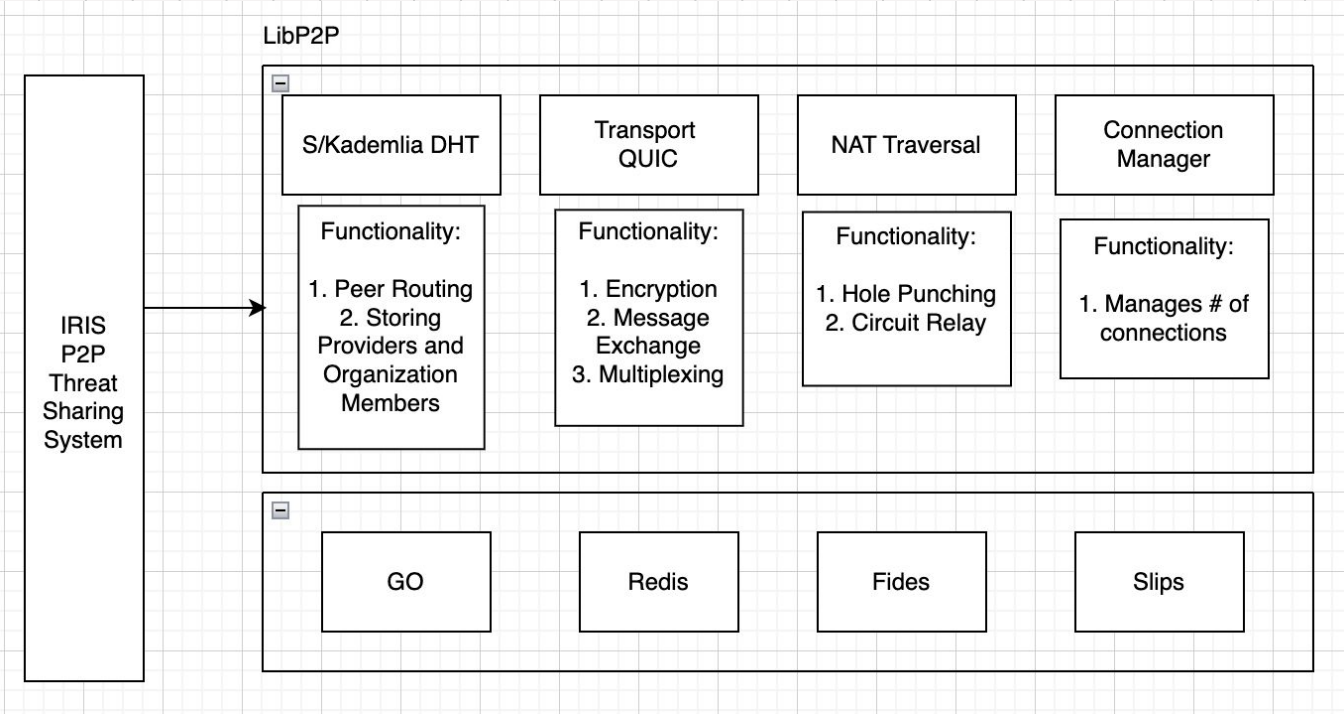
Feature Goals

Function 2:

- Requesting a Specific Chunk of File 1 (File 1.1)
  - Peer 2 looks in the DHT for peers with specific chunk
    - Peer 1 has File 1.1
  - Peer 2 requests File 1.1 from Peer 1
  - Peer 1 sends just File 1.1

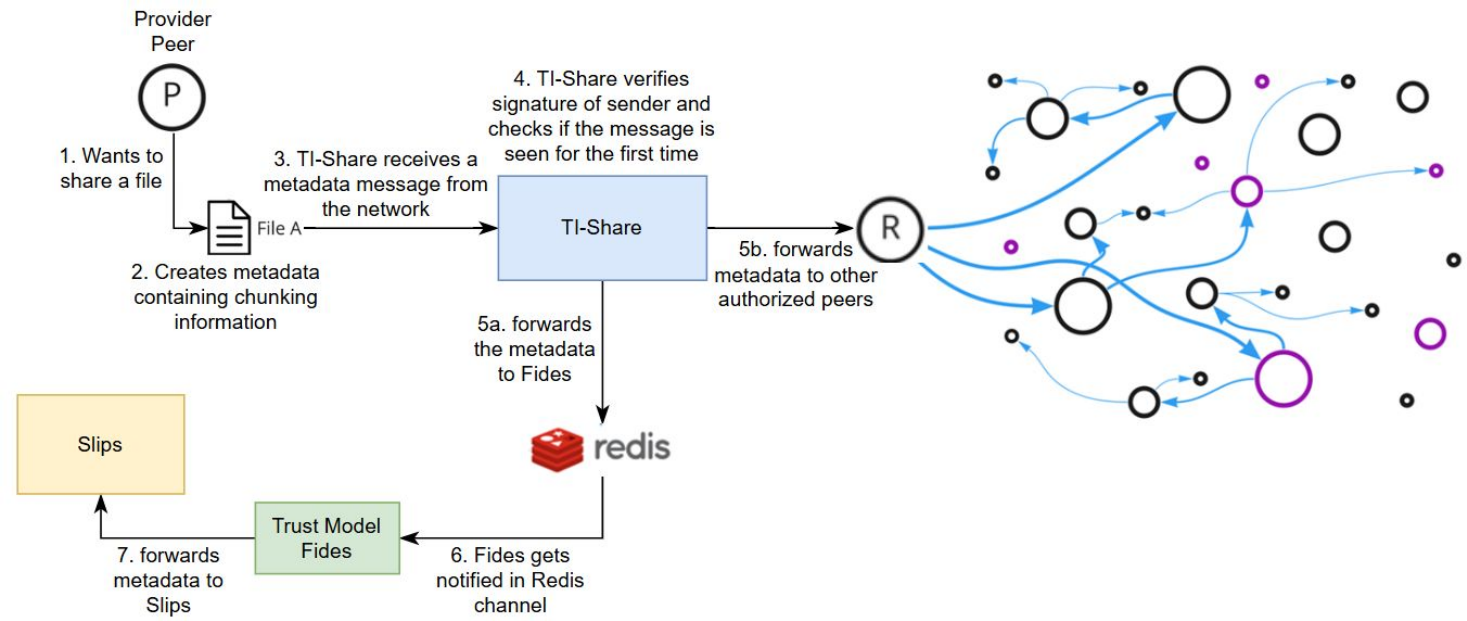


System Architecture



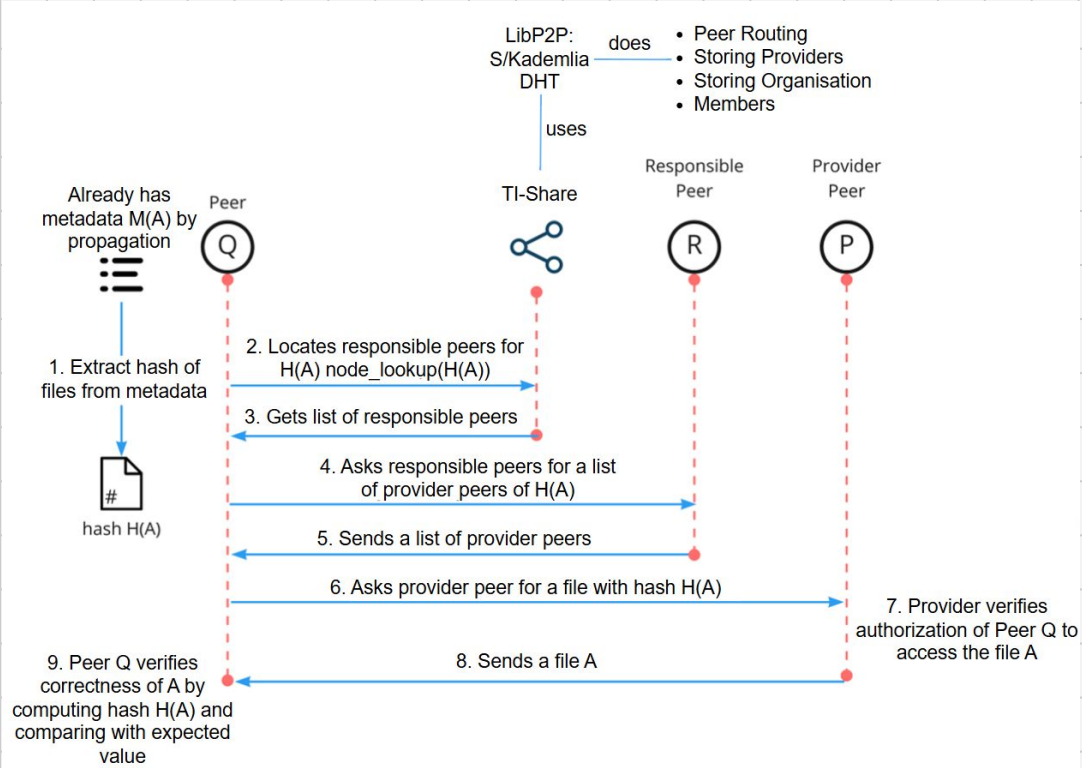
System Architecture

File Share Announce Walkthrough



System Architecture

File Download Walkthrough



Code Walkthrough

Structure of New Meta Data

```
type Nl2TlRedisFileShareMetadata struct {
    FileId          string             `json:"file_id"`
    Severity        string             `json:"severity"`
    Sender          utils.PeerMetadata `json:"sender"`
    Description     interface{}        `json:"description"`
    TotalSize       int64              `json:"total_size"`      // total file size in bytes
    ChunkSize       int32              `json:"chunk_size"`      // size of each chunk
    ChunkCount      int32              `json:"chunk_count"`     // total number of chunks
    AvailableChunks []int32            `json:"available_chunks"`// indices of chunks available
}
```

Chunking Files

```go
// splits a file into chunks and stores them in the chunk storage
func (fb *FileBook) SplitFileIntoChunks(path string, chunkSize int32, baseDir string) (*cid.Cid, error) {
    f, err := os.Open(path)
    if err != nil {
        return nil, err
    }
    defer f.Close()

    fileCid, err := GetFileCid(path)
    if err != nil {
        return nil, err
    }

    meta := &FileMeta{
        Path:         path,
        ChunkSize:    chunkSize,
        ChunkCount:   0,
        ChunksStatus: []bool{},
    }
```

```
    // Use the provided baseDir for chunk storage
    chunkDir := filepath.Join(baseDir, "chunks", fileCid.String())
    err = os.MkdirAll(chunkDir, os.ModePerm)
    if err != nil {
        return nil, err
    }

    buf := make([]byte, chunkSize)
    for {
        n, err := f.Read(buf)
        if err != nil && err != io.EOF {
            return nil, err
        }
        if n == 0 {
            break
        }

        chunkPath := filepath.Join(chunkDir, fmt.Sprintf("%d", meta.ChunkCount))
        err = ioutil.WriteFile(chunkPath, buf[:n], os.ModePerm)
        if err != nil {
            return nil, err
        }

        meta.ChunkCount++
        meta.ChunksStatus = append(meta.ChunksStatus, true)
    }

    err = fb.AddFile(fileCid, meta)
    if err != nil {
        return nil, err
    }

    return fileCid, nil
}
```

File Announcement

```
type Tl2NlRedisFileShareAnnounce struct {
    ExpiredAt       int64       `json:"expired_at"`
    Description     interface{} `json:"description"`
    Severity        string      `json:"severity"`
    Path            string      `json:"path"`
    Rights          []string    `json:"rights"`
    TotalSize       int64       `json:"total_size"`      // total file size in bytes
    ChunkSize       int32       `json:"chunk_size"`      // size of each chunk
    ChunkCount      int32       `json:"chunk_count"`     // total number of chunks
    AvailableChunks []int32     `json:"available_chunks"`// indices of chunks available
}
```

Download Requests

```go
func (fs *FileShareProtocol) onDownloadRequest(data []byte) {
    fileAnnouncement := Tl2NlRedisFileShareDownloadReq{}
    err := json.Unmarshal(data, &fileAnnouncement)
    if err != nil {
        log.Errorf("error unmarshalling Tl2NlRedisFileShareDownloadReq from redis: %s", err)
        return
    }
    log.Debug("received file download request message from TL")

    fileCid, err := cid.Decode(fileAnnouncement.FileId)
    if err != nil {
        log.Errorf("error decoding file cid: %s", err)
        return
    }
    meta := fs.fileBook.Get(&fileCid)
    if meta == nil {
        log.Errorf("file with cid %s has no stored metadata", fileCid.String())
        return
    }
    if meta.Available && meta.Path != "" {
        log.Errorf("file with cid %s is already available locally %s", fileCid.String(), meta.Path)
        return
    }
    // TODO shall I also check if I have rights for the file? Or can I assume that?
    providers, err := fs.dht.GetProvidersOf(fileCid)
    if err != nil {
        log.Errorf("error getting providers of file %s: %s", fileCid.String(), err)
        return
    }
    if len(providers) == 0 {
        log.Errorf("Found no providers of %s in DHT", fileCid.String())
        return
    }
    // sort providers based on their reliability to decreasing order
    fs.ReliabilitySort(providers)
```

```go
// if specific chunk indices are requested, handle chunked download
if len(fileAnnouncement.ChunkIndices) > 0 {
    // for each requested chunk, initiate a chunk download
    for _, chunkIndex := range fileAnnouncement.ChunkIndices {
        path, sender := fs.downloadSingleChunk(fileCid, chunkIndex)
        if path == "" || sender == nil {
            log.Errorf("failed to download chunk %d of file %s", chunkIndex, fileCid.String())
            continue
        }
        // notify TL about the downloaded chunk
        err = fs.notifyTLAboutDownload(fileCid, *sender, path, chunkIndex)
        if err != nil {
            log.Errorf("error sending download confirmation to redis for chunk %d: %s", chunkIndex, err)
            continue
        }
        log.Infof("successfully downloaded chunk %d of file %s", chunkIndex, fileCid.String())
    }
```

```
} else {
    // otherwise, proceed with the full file download as in original implementation
    // now use DHT to download the file
    path, sender := fs.downloadFile(providers, fileCid)
    if path == "" || sender == nil {
        // did not succeed
        return
    }
    // notify TL where file is downloaded
    err = fs.notifyTLAboutDownload(fileCid, *sender, path, -1)
    if err != nil {
        log.Errorf("error sending download confirmation to redis: %s", err)
        return
    }
    meta.Available = true
    meta.Path = path
    err = fs.dht.StartProviding(fileCid)
    if err != nil {
        log.Errorf("error starting providing file in DHT: %s", err)
    }
    log.Infof("successfully downloaded the file %s to path %s", fileCid.String(), path)
}
```

```
    // Check if all chunks are downloaded
    meta = fs.fileBook.Get(&fileCid)
    if meta != nil && meta.IsComplete() {
        outputPath := fmt.Sprintf("%s/%s", fs.downloadDir, fileCid.String())
        err := fs.fileBook.ReassembleFile(&fileCid, outputPath, fs.downloadDir)
        if err != nil {
            log.Errorf("error reassembling file %s: %s", fileCid.String(), err)
            return
        }
        log.Infof("successfully reassembled file %s to path %s", fileCid.String(), outputPath)
        meta.Available = true
        meta.Path = outputPath
    }
}
```

Initiate Downloading Single Chunk

```go
// helper function to download individual chunk
func (fs *FileShareProtocol) downloadSingleChunk(fileCid cid.Cid, chunkIndex int32) (string, *utils.PeerMetadata) {
    // create a FileDownloadRequest specifically for this chunk
    req := &pb.FileDownloadRequest{
        Cid:          fileCid.String(),
        ChunkSize:    fs.getChunkSizeFor(fileCid), // Define this helper to retrieve the chunk size from metadata
        ChunkIndices: []int32{chunkIndex},
    }
    path, err := fs.tryDownloadChunk(req, fileCid)
    if err != nil {
        log.Errorf("error downloading chunk %d: %s", chunkIndex, err)
        return "", nil
    }
    // return the local path where the chunk is stored and the peer metadata
    return path, fs.getLastSuccessfulProvider()
}
```

Downloading Single Chunk

```
// helper to download individual chunks by calling tryFileProvider
func (fs *FileShareProtocol) tryDownloadChunk(req *pb.FileDownloadRequest, fileCid cid.Cid) (string, error) {
    // get providers from the DHT for this file
    providers, err := fs.dht.GetProvidersOf(fileCid)
    if err != nil {
        return "", errors.Errorf("error getting providers of file %s: %s", fileCid.String(), err)
    }
    if len(providers) == 0 {
        return "", errors.Errorf("no providers found for file %s", fileCid.String())
    }
    // try each provider until one returns the chunk successfully
    for _, p := range providers {
        // use tryFileProvider to send the request/get the response
        path, err := fs.tryFileProvider(req, p.ID, fileCid)
        if err != nil {
            log.Error(err)
            continue
        }
```

```
        // If we successfully downloaded the chunk, update the ChunksStatus in the metadata
        if len(req.ChunkIndices) > 0 {
            meta := fs.fileBook.Get(&fileCid)
            if meta != nil {
                for _, chunkIndex := range req.ChunkIndices {
                    err = fs.fileBook.UpdateChunkStatus(&fileCid, chunkIndex, true)
                    if err != nil {
                        log.Errorf("error updating chunk status for chunk %d: %s", chunkIndex, err)
                        // Continue even if there's an error updating the status
                    } else {
                        log.Debugf("updated chunk status for chunk %d of file %s", chunkIndex, fileCid.String())
                    }
                }
            } else {
                log.Errorf("metadata not found for file %s after download", fileCid.String())
            }
        }

        // If we get a valid path (i.e. the chunk was downloaded), return it.
        return path, nil
    }
    return "", errors.Errorf("failed to download chunk from all providers for file %s", fileCid.String())
}
```

Reassemble File

```go
// reassembles a file from its chunks and writes it to the output path string)
func (fb *FileBook) ReassembleFile(cid *cid.Cid, outputPath string) error {
    meta := fb.Get(cid)
    if meta == nil {
        return errors.Errorf("file with cid %s not found", cid.String())
    }

    // Ensure all chunks are available
    if (!meta.IsComplete()) {
        return errors.Errorf("not all chunks are available for file %s", cid.String())
    }

    // Open the output file for writing
    outputFile, err := os.Create(outputPath)
    if err != nil {
        return err
    }
    defer outputFile.Close()

    // Read and concatenate chunks in orderString())
    chunkDir := filepath.Join("chunks", cid.String())
    for i := int32(0); i < meta.ChunkCount; i++ {
        chunkPath := filepath.Join(chunkDir, fmt.Sprintf("%d", i))
        chunkData, err := os.ReadFile(chunkPath)
        }
        _, err = outputFile.Write(chunkData)
        if err != nil {File.Write(chunkData)
            return errors.Errorf("error writing chunk %d to output file for file %s: %s", i, cid.String(), err)
        }return errors.Errorf("error writing chunk %d to output file for file %s: %s", i, cid.String(), err)

    // Update metadata to mark the file as reassembled
    meta.Available = true mark the file as reassembled
    meta.Path = outputPath
    meta.Path = outputPath
    return nil
}
```

Demo

Peer Interaction Functionality

Alert

- PUBLISH gp2p_tl2nl1 '{"type": "<>", "version": <>, "data": { "payload": "<>" }}'

Recommendation REQUEST

- PUBLISH gp2p_tl2nl1 '{"type": "<>", "version": <>, "data": { "<>": ["<>"], "payload": "<>" }}'

Recommendation RESPONSE

- PUBLISH gp2p_tl2nl0 '{"type": "<>", "version": <>, "data": {"request_id": "<>", "recipient_id": "<>", "payload": "<>"}}'

Intelligence REQUEST

- PUBLISH gp2p_tl2nl1 '{"type": "<>", "version": <>, "data": {"payload": "<>"}}'

Intelligence RESPONSE

- PUBLISH gp2p_tl2nl2 '{"type": "<>", "version": <>, "data": {"request_id": "<>", "payload": "<>"}}'

FileShareAnnounce

- PUBLISH gp2p_tl2nl2 '{"type": "<>", "version": <>, "data": { "expired_at": <>, "severity": "<>", "rights": [], "description": {"size": <>}, "path": "<>" }}'

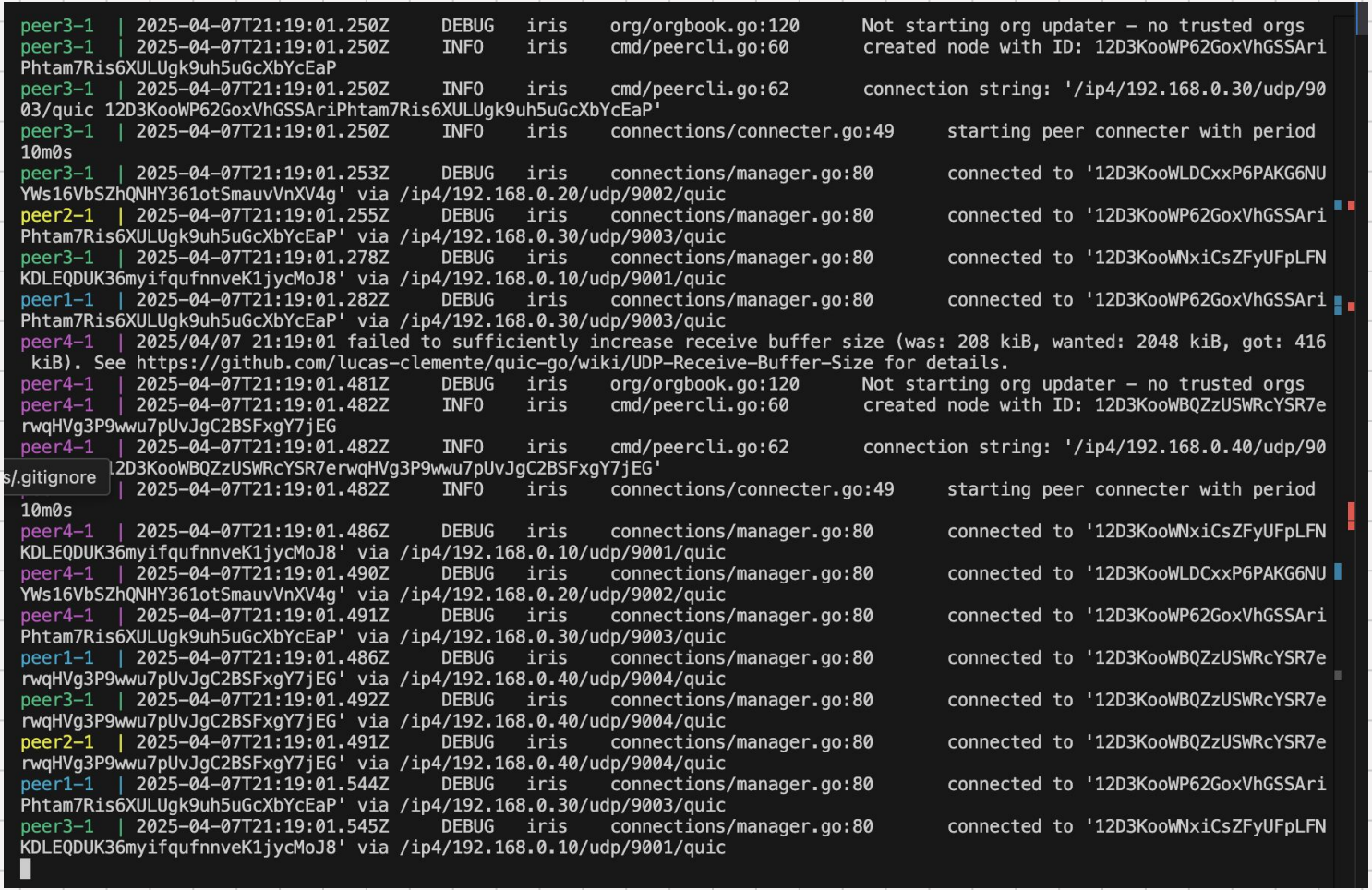
FileShareDownload

- PUBLISH gp2p_tl2nl1 '{"type": "<>", "version": <>, "data": {"file_id": "<>" }}'

Send Reliability Update from TL to NL

- PUBLISH gp2p_tl2nl6 '{"type": "<>", "version": <>, "data": [{"peer_id": "<>", "reliability": <>}]}'

Network for 4 Peers on different subnets



4 containers

```
ring_2025/IntSys/Project/iris$ docker-compose up -d
WARN[0000] /mnt/c/Users/Maria/Documents/Desktop/College/Spring_2025/
IntSys/Project/iris/docker-compose.yml: the attribute `version` is o
bsolete, it will be ignored, please remove it to avoid potential con
fusion
[+] Running 5/5
 ✔Container iris-redis-1  Started                              0.6s
 ✔Container iris-peer1-1  Started                              1.0s
 ✔Container iris-peer2-1  Started                              1.4s
 ✔Container iris-peer3-1  Started                              1.8s
 ✔Container iris-peer4-1  Started                              2.2s
```

File Share Announce

```
ring_2025/IntSys/Project/iris$ docker exec -it iris-peer1-1 /bin/bas
h
root@1fb07562e1db:/usr/iris# apt-get update
```

```
root@1fb07562e1db:/usr/iris# redis-cli -h iris-redis-1 -p 6379
iris-redis-1:6379> PUBLISH gp2p_tl2nl1 '{"type": "tl2nl_file_share",
 "version": 1, "data": { "expired_at": 1647162651, "severity": "MAJO
R", "rights": [], "description": {"size": 68}, "path": "/usr/iris/vo
l/key.priv" }}'
(integer) 1
```

File Share Announce

```
ring_2025/IntSys/Project/iris$ docker exec -it iris-peer2-1 /bin/bas
h
```

```
root@e829bf095df2:/usr/iris# redis-cli -h iris-redis-1 -p 6379
iris-redis-1:6379> SUBSCRIBE gp2p_tl2nl1
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "gp2p_tl2nl1"
3) (integer) 1
1) "message"
2) "gp2p_tl2nl1"
3) "{\"type\":\"nl2tl_file_share_received_metadata\",\"version\":1,\
"data\":{\"file_id\":\"QmS4FkBx1uBDHDLASvDocmfo5FXrXgNv4F8WRDkiNTUFe
7\",\"severity\":\"MAJOR\",\"sender\":{\"id\":\"12D3KooWLDCxxP6PAKG6
NUYWs16VbSZhQNHY361otSmauvVnXV4g\",\"organisations\":[]},\"descripti
on\":{\"size\":420}}}"
```

File Share Announce Demo

https://youtu.be/PbQt6o_Otws

File Sharing Demo

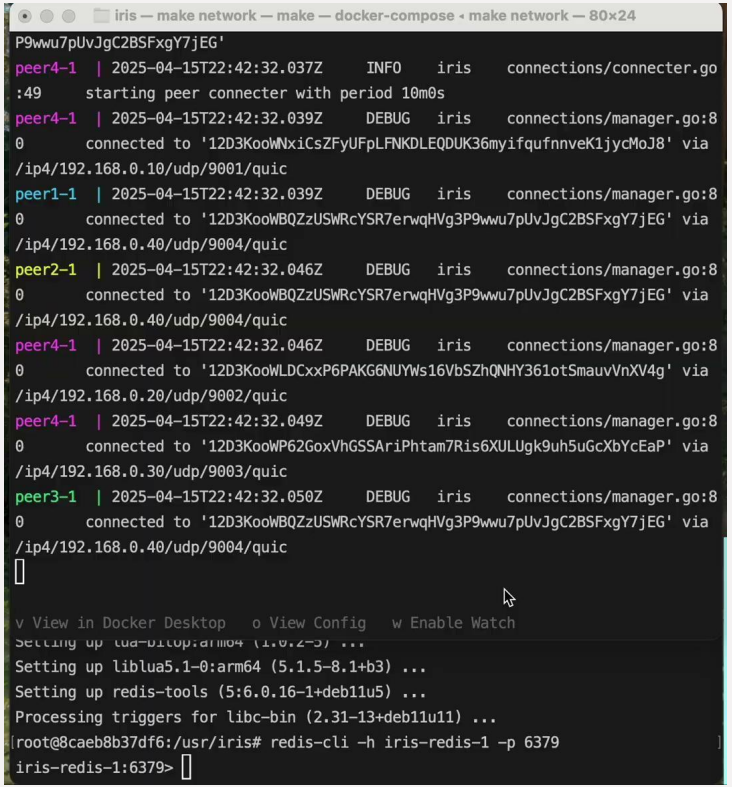
FileShareDemo.mp4

File Sharing Result

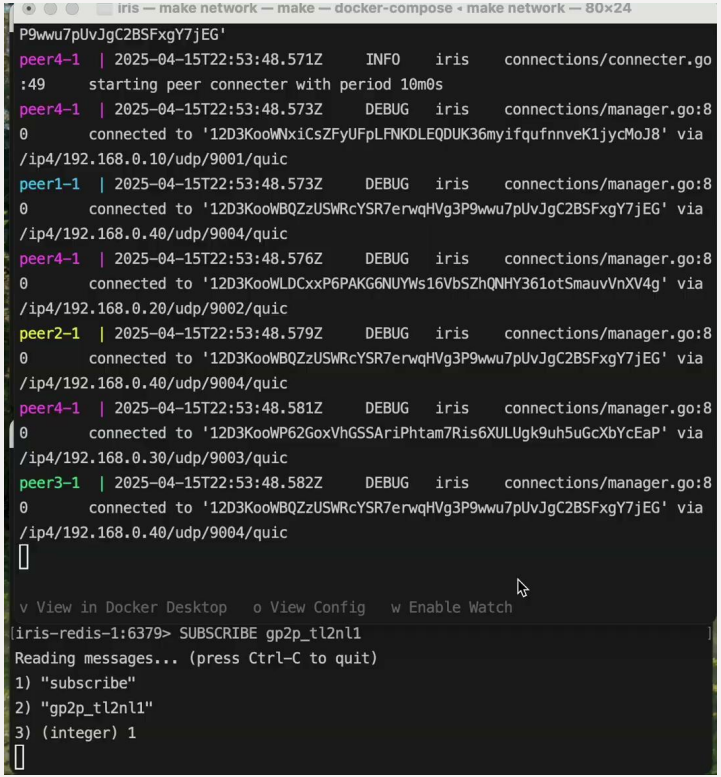
```
2025-04-18 14:12:35 2025-04-18T18:12:35.191Z    DEBUG   iris    protocols/fileshare.go:85       received file download request message from TL
2025-04-18 14:12:35 2025-04-18T18:12:35.195Z    DEBUG   iris    protocols/fileshare.go:155      trying to download QmS4FkBx1uBDHDLASvDocmfo5FXrXg
Nv4F8WRDkiNTUFe7 from 12D3KooWLDCxxP6PAKG6NUYWs16VbSZhQNHY361otSmauvVnXV4g
2025-04-18 14:12:35 2025-04-18T18:12:35.197Z    INFO    iris    protocols/fileshare.go:132      successfully downloaded the file QmS4FkBx1uBDHDLA
SvDocmfo5FXrXgNv4F8WRDkiNTUFe7 to path /tmp/QmS4FkBx1uBDHDLASvDocmfo5FXrXgNv4F8WRDkiNTUFe7
```

home			8 months ago	drwxr-xr-x
lib			12 days ago	drwxr-xr-x
lib64			12 days ago	drwxr-xr-x
media			12 days ago	drwxr-xr-x
mnt			12 days ago	drwxr-xr-x
opt			12 days ago	drwxr-xr-x
proc			24 minutes ago	dr-xr-xr-x
root	MODIFIED		43 minutes ago	drwx------
run			12 days ago	drwxr-xr-x
sbin			12 days ago	drwxr-xr-x
srv			12 days ago	drwxr-xr-x
sys			24 minutes ago	dr-xr-xr-x
tmp	MODIFIED		2 minutes ago	dtrwxrwxrw:
QmS4FkBx1uBDHDLASvDocmfo5FXrXgNv4F8WRDkiNTUFe7	ADDED	571 Bytes	2 minutes ago	-rw-r--r--
usr	MODIFIED		10 days ago	drwxr-xr-x
var	MODIFIED		12 days ago	drwxr-xr-x

File Transfer Partial



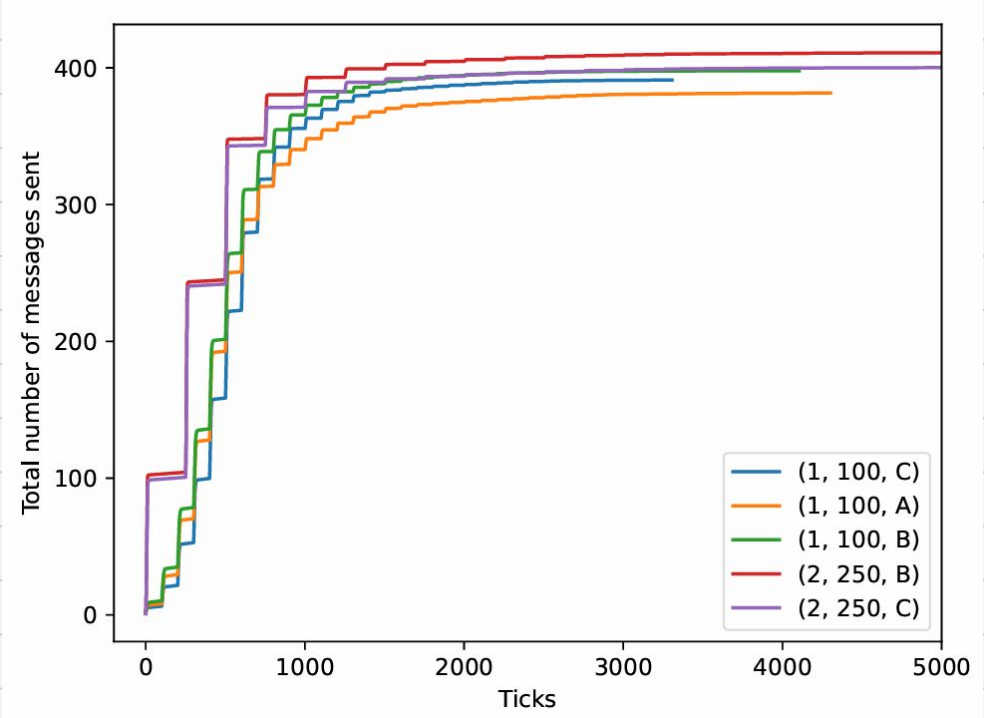
File Transfer Full



Evaluation

IRIS Metrics

- **Epidemic Protocol**
- **Transfer of file metadata: 300 ms**
- **To account for low bandwidth and latency converted to 1 tick.**
- **Tick (500 ms): time period in which peers may transfer one gossip to peers they share an edge with.**



Easy vs Hard Case

Metadata message has:

Normal Severity

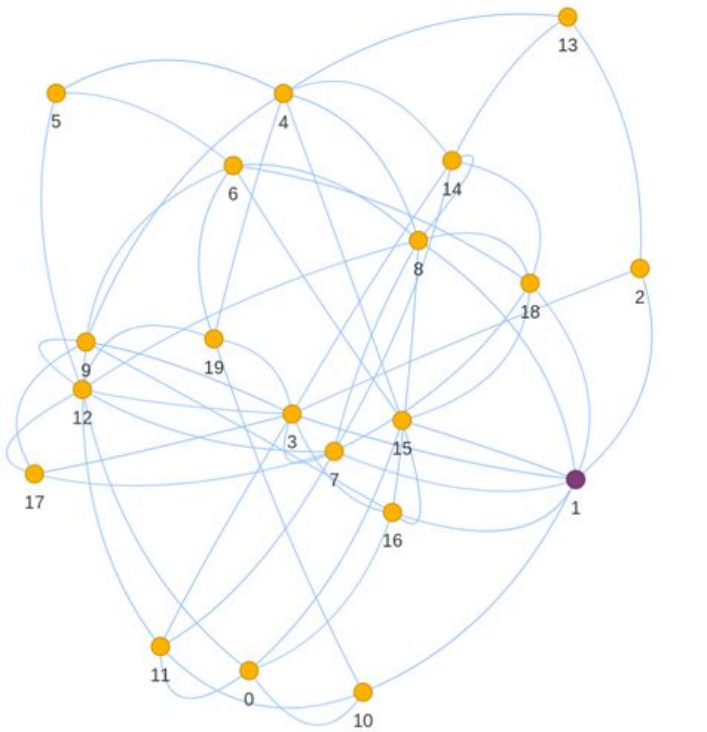
Critical Severity

Experiment

<u>Purpose</u>: to select optimal spreading strategy for spreading file metadata with different levels of severity.

The Spreading Strategy is defined by/a permutation of:

1. **Spreading Factor - how many nodes an infected node should spread a message to at once. {1,2,3,5,7,9, ALL possible recipients}**
2. **Spreading Period - how long an infected node should wait until they ask/spread again.**
3. **Algorithm to choose recipients (uniform probability, biggest service trust, probability that exponentially grows with service trust)**



Network of 20 peers. Purple node is randomly selected to spread a message given spreading strategy.

Experiment

These differing strategies are run in 4 scenarios defined by the ratio of malicious peers in the network (0%, 25%, 50%, 75%).

Malicious Peers:

- **In reality: They make various attacks to the network that prevent benign peers from getting the threat information they need. Ex: A malicious peer keeps making file share announcements causing another peer to miss a critical announcement among the now flooded network.**
- **In this experiment: The behavior of all peers is the same. There is no attack modeled. This is due to use of Fides which assigns malicious peers a smaller service trust due to their past malicious behavior. So for this experiment malicious peers are defined as those with smaller service trust. The inclusion of them is to see which, if any spreading strategy favor spreading announcements to malicious peers over benign peers when it shouldn't.**

Experiment

Ending simulation:

- **All peers begin in *Susceptible* state. The peer spreading the message is in the *Infected* state. When a peer received the gossip it moves to the Infected State. If an Infected peer has sent the gossip to their list of candidates for sharing gossip they are moved to the *Removed* state. Once no peers are in the Susceptible state, the simulation will end.**
- **Spreading expiration (the max number of allowed ticks) is also set to 10,000 if above does not occur.**

Metrics without Malicious Peers

Metric to check for flooding: repeated messages per tick

Metric to check speed of spreading: average duration of spreading

The Figure represents:

- **5 best and 5 worst spreading strategies in terms of repeated messages and average duration of spreading per tick in environment with no malicious peers.**

spreading strategy			repeated messages per tick	ratio of repeated messages	average duration of spreading	worst case duration of spreading
factor	period	choosing recipients				
1	100	C	0.33	0.69	7.1min	45.86min
1	100	A	0.35	0.68	6.75min	29.21min
1	100	B	0.35	0.70	7.05min	45.08min
2	250	B	0.43	0.71	6.96min	41.73min
2	250	C	0.44	0.70	6.68min	56.32min
...						
9	1	A	106.66	0.78	1.93s	6.5s
9	1	B	106.97	0.78	2.035s	11.0s
ALL	1	B	112.72	0.78	1.93s	10.5s
ALL	1	C	112.84	0.78	1.89s	9.0s
ALL	1	A	112.84	0.78	1.86s	9.5s

Metrics with Malicious Peers

This Figure represents:

- **5 best spreading strategies in terms of repeated messages in an environment with malicious peers.**
- **5 worst spreading strategies in terms of average duration of spreading to benign peers in an environment with malicious peers.**

ratio of malicious peers	spreading strategy			repeated messages per tick		ratio of repeated messages		average duration of spreading		worst case duration of spreading	
	factor	period	choosing recipients	all peers	benign peers	all peers	benign peers	all peers	benign peers	all peers	benign peers
25%	1	100	A	0.30	0.21	0.68	0.66	7.92min	7.64min	> 85min	49.23min
	1	100	C	0.30	0.21	0.69	0.68	7.85min	7.21min	> 85min	39.18min
	1	100	B	0.33	0.25	0.70	0.70	7.47min	6.25min	> 85min	50.85min
	2	250	C	0.38	0.27	0.70	0.68	7.38min	6.62min	> 85min	56.27min
	2	250	A	0.41	0.29	0.69	0.67	7.71min	7.33min	> 85min	77.1min
50%	1	100	A	0.23	0.10	0.68	0.63	11.09min	10.01min	> 85min	77.57min
	1	100	C	0.24	0.11	0.68	0.65	10.28min	7.97min	> 85min	80.07min
	1	100	B	0.26	0.13	0.69	0.68	9.5min	6.29min	> 85min	56.75min
	2	250	C	0.30	0.13	0.69	0.65	9.85min	7.41min	> 85min	66.72min
	2	250	A	0.30	0.13	0.69	0.64	11.07min	9.85min	> 85min	70.88min
75%	1	50	C	0.36	0.08	0.68	0.60	9.0min	5.13min	> 85min	59.62min
	2	100	C	0.51	0.11	0.68	0.60	7.99min	4.11min	> 85min	77.53min
	3	250	C	0.46	0.11	0.69	0.62	11.37min	5.43min	> 85min	77.1min
	2	100	B	0.49	0.13	0.69	0.64	8.11min	3.34min	> 85min	57.52min
	1	20	A	0.85	0.17	0.68	0.56	5.1min	3.55min	> 85min	79.55min

Metrics with Malicious Peers

This Figure represents:

- **5 best spreading strategies in terms of average duration of spreading to benign peers in an environment with malicious peers.**
- **5 worst spreading strategies in terms of repeated messages in an environment with malicious peers.**

ratio of malicious peers	spreading strategy			repeated messages per tick		ratio of repeated messages		average duration of spreading		worst case duration of spreading	
	factor	period	choosing recipients	all peers	benign peers	all peers	benign peers	all peers	benign peers	all peers	benign peers
25%	9	1	B	102.74	79.02	0.77	0.77	2.025s	1.845s	> 85min	8.0s
	9	1	C	103.81	76.19	0.77	0.76	1.95s	1.86s	> 85min	5.0s
	ALL	2	B	97.77	75.16	0.77	0.77	2.105s	1.925s	> 85min	7.5s
	ALL	1	A	102.40	74.36	0.77	0.76	1.995s	1.95s	> 85min	13.0s
	ALL	1	C	96.58	70.77	0.77	0.76	2.075s	1.98s	> 85min	6.5s
50%	9	1	B	79.74	41.06	0.76	0.76	2.635s	2.01s	> 85min	9.5s
	9	2	B	84.05	43.46	0.76	0.76	2.71s	2.02s	> 85min	28.0s
	9	2	C	84.29	40.27	0.75	0.74	2.495s	2.055s	> 85min	33.0s
	ALL	2	B	66.13	35.33	0.75	0.75	3.555s	2.195s	> 85min	15.5s
	ALL	2	B	72.22	37.11	0.76	0.76	2.8s	2.205s	> 85min	23.0s
75%	ALL	1	B	65.47	17.48	0.74	0.72	5.555s	2.17s	> 85min	44.0s
	9	1	C	64.62	15.57	0.74	0.70	5.6s	2.27s	> 85min	24.5s
	ALL	2	B	61.19	16.22	0.74	0.72	6.205s	2.285s	> 85min	36.0s
	9	1	C	61.33	14.67	0.74	0.70	5.205s	2.285s	> 85min	23.0s
	7	1	B	55.97	14.72	0.74	0.72	5.825s	2.365s	> 85min	26.0s

Results

Findings: Regardless the number of malicious peers in the networks we got

Fastest Coverage:

- ↑ **Spreading Factor**
- ↓ **Spreading Period**

Using a spreading strategy with a high spreading factor and low spreading period is ideal for our hard case when the metadata message has critical severity which means it must be sent as fast as possible to their recipients even at the cost of flooding the network with messages.

Minimal Flooding:

- ↓ **Spreading Factor**
- ↑ **Spreading Period**

Using a spreading strategy with low spreading factor and high spreading period is ideal for our easy case when the metadata message has normal severity which means the message doesn't need to spread as fast so it should try to minimize flooding in the network.

Results

Findings: Impact of Malicious Peers

Algorithm for choosing recipient:

- **no substantial effect in moderating the level of flooding the networks**
- **except in environment with 75% malicious peers where the best strategy was to choose recipients with a probability that exponentially grows with candidates' service trust.**
- **Conclusion: Assuming higher adversarial network, using service trust is a reliable heuristic to choose message recipients.**

As shown on the right worst case duration of spreading grows proportionally with maliciousness of the network.

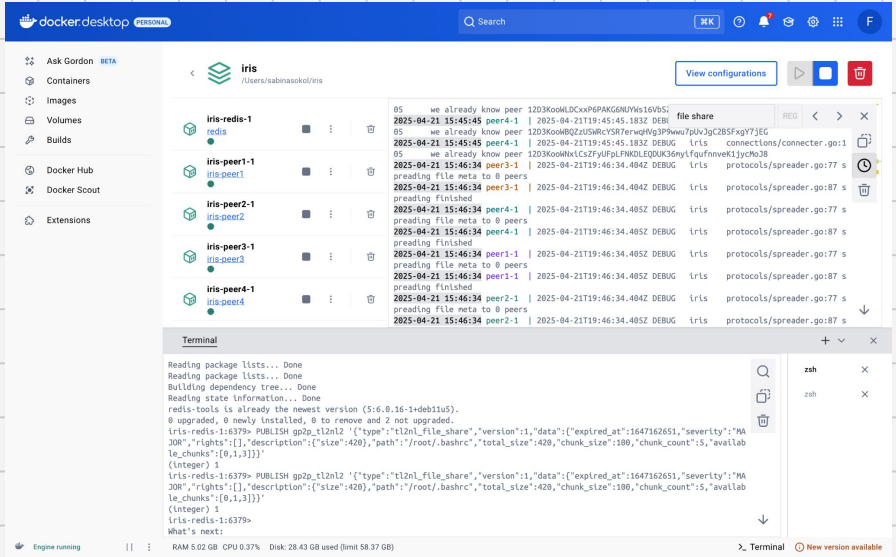
ratio of malicious peers	worst case duration of spreading	
	all peers	benign peers
25%	> 85min	8.0s
	> 85min	5.0s
	> 85min	7.5s
	> 85min	13.0s
	> 85min	6.5s
50%	> 85min	9.5s
	> 85min	28.0s
	> 85min	33.0s
	> 85min	15.5s
	> 85min	23.0s
75%	> 85min	44.0s
	> 85min	24.5s
	> 85min	36.0s
	> 85min	23.0s
	> 85min	26.0s

Query Comparisons Method

4 peer network, **Peer 1** shares the following file meta data announcement:

PUBLISH gp2p_tl2nl2 '{"type":"tl2nl_file_share","version":1,"data":{"expired_at":1647162651,"severity":"MAJOR","rights":[],"description":{"size":420},"path":"/root/.bashrc","total_size":420,"chunk_size":100,"chunk_count":5,"available_chunks":[0,1,3]}}'

Calculated time from file meta data announcement to **successful spreading and received** by all 4 peers as well as **hops** based on Docker logs



Query Comparisons Results

Team Member	Spreading/Received Time	Hops
Sabina Sokol	8 ms	1 -> 2 -> 4 -> 3
Keerthana Thotakura	6 ms	1-> 4 -> 2 -> 3
Maria Jothish	10 ms	1 -> 3 -> 4 -> 2
Tran Ha	5 ms	1 -> 2 -> 4 -> 3
Seung-a Baek	9 ms	1 -> 3 -> 2 -> 4

Sabina’s Analysis

My query took 8 ms to spread the file meta data from peer 1 to the rest of the peers in the network.

IRIS uses go-libp2p as the message propagation/sharing framework and relies on Gossipsub, which is a probabilistic pub-sub protocol that involves randomization in peer selection and message forwarding intervals. For our file meta data announcement, each peer gossips metadata to a subset of its peers, who in turn gossip further. Even with only 4 peers, peer-to-peer gossiping isn’t strictly deterministic — different relaying paths may be taken on different runs. Thus, the nature of the message sharing introduces small latency variability.

Also, since IRIS uses QUIC and runs in 4 Docker containers alongside Redis, metadata spreading experiences natural variability due to virtual networking jitter and Docker container scheduling delays. Redis traffic competing for CPU or I/O can also briefly delay libp2p message handling. Together, these can cause 5–10 ms variation in how quickly file metadata announcements propagate across peers in repeated runs.

Keerthana’s Analysis

For my query the latency between the file share announcement being shared by peer 1 to all of the peers receiving the announcement was 6 milliseconds. The average latency for all the queries was approximately 7.6 milliseconds which shows that my query ran somewhat faster than the other queries. Additionally, if we analyze the path between the peers that the query took, mine was somewhat unique as it went from peer 1 to peer 4 to peer 2 and then to peer 3 which was different from all the other peers. The faster speed of my query in comparison to the average and the majority of the other queries could be correlated to the path that the query took and it being a more optimal path. Additionally there could be some minute processing time differences or natural variability in the system that could have caused the discrepancy and my lower latency.

Maria’s Analysis

The stats I got when running the file share announce query was 10 ms and the path it took for spreading from peer 1 was 1->3->4->2. I believe the variation in spread times was caused by how the network was initially set up. If you investigate the setup of each of the peers, you will see this:

	Connection String	Peer Discovery (List of multi addresses)
Peer 1	/ip4/192.168.0.10/udp/9001/quic …	"/ip4/192.168.0.30/udp/9003/quic …”
Peer 2	/ip4/192.168.0.20/udp/9002/quic …	"/ip4/192.168.0.10/udp/9001/quic …”
Peer 3	/ip4/192.168.0.30/udp/9003/quic …	"/ip4/192.168.0.20/udp/9002/quic …”
Peer 4	/ip4/192.168.0.40/udp/9004/quic …	"/ip4/192.168.0.10/udp/9001/quic …” "/ip4/192.168.0.20/udp/9002/quic …” "/ip4/192.168.0.30/udp/9003/quic …”

Based on this we can identify the initial connectivity of the network. Peer 1 only knows peer 3, peer 2 only knows peer 1, peer 3 only knows peer 2, and peer 4 knows peers 1,2, and 3. From this point peers may choose to a) initiate connections themselves (e.g. peer 1 makes itself known to peer 3), b) share information on its known peers (e.g. peer 3 tells peer 1 about peer 2), c) peers may look up each other in the DHT once the network is made. After this when we make the file share announcement the actual connectivity between peers is unknown and varies based on the time between making the network and then sending the file share announcement. Given my result and Seung-a’s, it seems as though peer 1 takes quite some time making itself known to peer 3 when spreading the metadata or peer 3 takes a long time to make itself known to another peer and then process and send metadata. The likelihood is the latter because in the cases with the smallest spreading time (Keerthana’s and Tran’s) peer 3 didn’t do any spreading.

Another general cause for the slight variation in spread times despite having the same path are network latency delays causing peers to receive the message later or take longer to act on a received message.

Tran’s Analysis

For my query, the file metadata announcement took 5 ms, which was the fastest time in our group. The metadata traveled along the path of peer 1 -> peer 2 -> peer 4 -> peer 3, and it seem to be the most efficient. This is interesting because the other path in the test where peer 3 was involved more early on, it took longer compare to when peer 3 is at the end. This seem to suggest that peer 3 may sometime cause a slight delay during the process. Still, these difference is almost negligible, all the spreading time was ranged from 5 ms to 10 ms, demonstrate that the system is capable of distributing the metadata very quickly in the network. This suggest that the underlying communication mechanism and Docker-based virtual environment introduce minimal latency, which is critical for real-time threat intelligence or file-sharing scenarios. And since it is decentralized, it allows the metadata to traverse different path across the network. Yet, all queries were able to be consistent, which shows its reliability.

Seung-a’s Analysis

My query took 9ms while others reported times ranging from 5ms to 10ms. While a minute difference in integer values, the variation in runtimes comes from the non-deterministic nature of peer-to-peer gossip, virtualized environments, and the shared-resource scheduling behavior inside each of the 4 containers. My 9ms query compared to a 5ms query can be attributed to a less optimal path chosen for my specific run.

My query took the path from peer 1 to peer 3 to peer 2 then to peer 4. The fastest path of 5ms was 1 -> 2 -> 4 -> 3 while my longest path of 10ms was 1 -> 3 -> 4 -> 2. All paths started from peer 1 while my path which took the second longest and the longest path both went from peer 1 to peer 3. From this, building off of Maria’s analysis, we can make the assumption that peer 3 may have a difficult time making itself known to other peers and sending metadata. All other query paths finished at peer 3.

Final Notes

Novelty

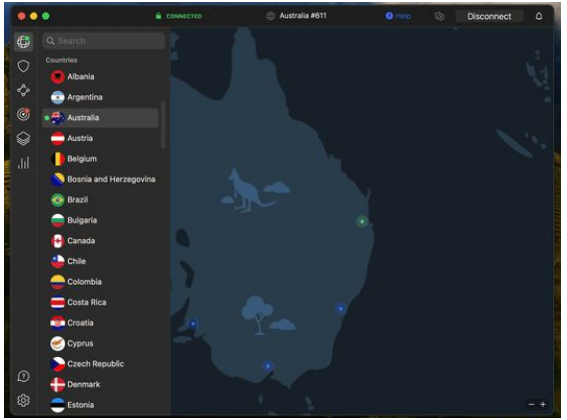
Feature / Capability	Traditional Systems (e.g., MISP, STIX/TAXII, centralized feeds)	TI-SHARE (Decentralized + Chunked P2P)
Architecture	Centralized or federated servers	Fully decentralized peer-to-peer mesh
Data Dissemination	Bulk file transfers, full feed ingestion	Chunked, selective file retrieval via libp2p
Anonymity / Privacy	Pseudonymization or shared trust anchors	Network-level anonymity via libp2p
Real-Time Distribution	Often batch or push-pull (delayed)	Real-time pubsub + background chunk sync
Scalability	Bottlenecks at central nodes or trust authorities	Scales organically with peers and interest-driven gossip

Strengths and Weaknesses

STRENGTH

Requires Sending/Exchanging Larger File Meta Data, But No Noticeable Difference

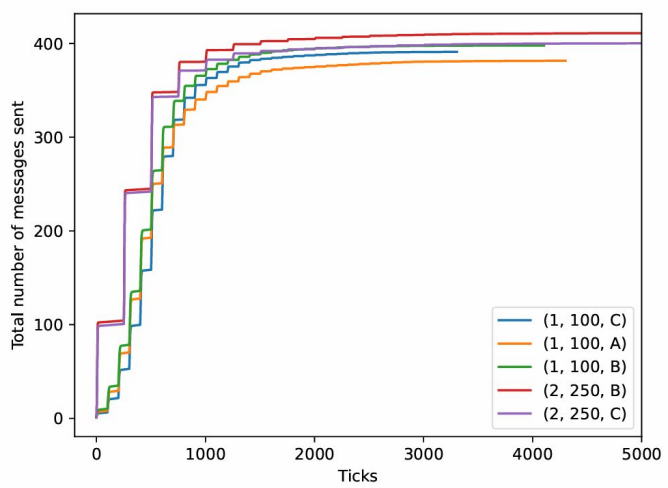
- Modeled after IRIS evaluation metrics by using NordVPN to deploy one peer in Australia and one peer locally
- File transfer (one way message) still took approximately **300 ms** because file meta data only increased by **~60 bytes**, which is negligible
- Original assumptions of **1 tick = 500 ms** still holds -> same statistics as previously



WEAKNESS

File Reassembly Overhead/Latency

- Average file is **5 MB**, split into **256 KB** chunks for a total of **20 chunks**
- Reassembly requires in-memory copying/concatenation, which takes roughly **10 ms** assuming that all chunks are available
- Does not include additional overhead for file-level integrity verification



Problems Encountered

Original IRIS Codebase Was Not Future Proofed

- Tradeoffs between upgrading existing implementation versus enhancing implementation with novel features

```
 => ERROR [peer1 build 8/8] RUN go build cmd/peercli.go                                                  23.2s
------
 > [peer1 build 8/8] RUN go build cmd/peercli.go:
17.43 # github.com/lucas-clemente/quic-go/internal/qtls
17.43 /go/pkg/mod/github.com/lucas-clemente/quic-go@v0.24.0/internal/qtls/go118.go:6:13: cannot use "quic-go doesn't build on Go
 1.18 yet." (untyped string constant) as int value in variable declaration
```

Very Limited Documentation on IRIS Useability

- Significant time was spent troubleshooting file announcements, file download confirmations, etc.
- Updated documentation for file sharing and file announcement features for future ease of use

Future Directions

- **We have yet to simulate a malicious actor tying to exploit the epidemic protocol by disseminating a large number of messages. This could result in flooding the network and eventually leading to DDoS of the entire network.**
- **A potential solution we are looking into is rate-limiting per peer where every peer keeps track of the number of sent messages by all neighbors and only allows a certain number of them per unit of time.**

Thank you!

Q&A

References

IRIS

Short description: https://nlnet.nl/project/Iris-P2P/

Github: https://github.com/stratosphereips/iris/tree/main?tab=readme-ov-file