Pike: Making SMB Testing Less Torturous

Brian Koropoff
EMC Isilon

http://github.com/emc-isilon/pike

What is Pike?

- A framework for writing SMB2/SMB3 protocol correctness tests
- Written in (almost) pure Python

Demo

What's Available Today

- Support for a modest subset of SMB2/3
  - Currently more depth than breadth
  - Emphasis on fiddly cases like SMB3 failover, complex create contexts, leases, ...
- Modest set of tests
- Open Source under a BSD license
- Targeting Linux, FreeBSD
- Great platform for future work

Motivation

- Existing mature solutions largely in C
  - Not convenient for rapid prototyping
  - Writing bindings for another language painful
- Many implementations emphasize being a client
  - Abstract away details that need to be tweaked or inspected for protocol conformance testing

Why Python?

- Ubiquitous
- Expressive
- Flexible
- Huge ecosystem to draw upon

Design Goals

- Emphasize flexibility and ease-of-use over performance
  - Correctness testing is the focus, load testing would be a bonus
- Provide convenient abstractions but allow bypassing them
  - Simple cases should be simple, hard cases should be possible
- Be extensible, reusable

Overview

- How Pike is architected
- How to extend it
- How to use it

Architecture

- Core Primitives
  - Abstract data model, (de)serialization support
- SMB2/3 packet definitions
  - What the protocol looks like on the wire
- SMB2/3 client model
  - Connection and state tracking
  - Request/response processing
- Test harness

Core Primitives (pike.core)

- Cursor
  - Data encoding/decoding
- Frame
  - Hierarchical packet model
- Enums
  - Protocol constants
- Anti-boilerplate magic

Cursor

- (buffer, offset) pair indicating read/write location
- Manipulate offset in place with +=, -=
- Return new derived cursor with copy(), +, -
- Take the difference of two cursors with -
- Encode/decode functions (integers, strings, …)
- Scoped bounds checking
- Holes: placeholders for backpatching
  - Offset and length fields, checksums, …

Basic Cursor Usage


```
# Create cursor pointing at start of empty buffer
cur = Cursor(array('B', []), 0)
# Encode some data into it
cur.encode_uint16le(42)
cur.encode_utf16le("Hello, world!")

# Rewind and read back data
cur.offset = 0
foo = cur.decode_uint16le() # 42
# Decode next 26 bytes as utf16le
bar = cur.decode_utf16le(26) # u"Hello, world!"
```

Using Bounds


```
def decode_something(cur):
    # Read length
    foo_len = cur.decode_uint32le()

    # Raise an exception if decode_foo() attempts
    # to read data outside of (cur, cur + foo_len]
    # The bounds only apply to the body of the with block
    with cur.bounded(cur, cur + foo_len):
        foo = decode_foo(cur)
    return foo
```

Using Holes


```
def encode_something(cur, foo):
    # We need to write out the length of the encoding of foo as a header,
    # but we don't know it yet.  Use a hole as a placeholder.
    len_hole = cur.hole.encode_uint32le(0)

    # Save the current cursor for later
    start = cur.copy()

    # Encode foo
    encode_foo(cur, foo)

    # Now we can fill the hole with the correct length
    len_hole(cur - start)
```

Frame

- Abstract segment of a packet
- Packet fields are simple attributes
- Can contain sub-frames as children
- Parent frame reachable via `parent` attribute
- Children reachable via indexing, iterating

Using Frames


```
# Instantiate a frame
foo = Foo(...)

# Set/get fields as attributes
foo.some_field = 5

# Add a child frame
bar = Bar(foo)

# Access children by index
assert foo[0] is bar

# Iterate children
for child in foo:
    assert child.parent is foo
    # Frames can be pretty-printed
    print child
```

Subclassing Frame


```
class Foo(pike.core.Frame):
    def __init__(self, parent):
        super(Foo, self).__init__(parent)
        # Set fields to default values in logical order
        self.some_field = 23
        # Attributes starting with _ are implementation details
        self._bars = []
```

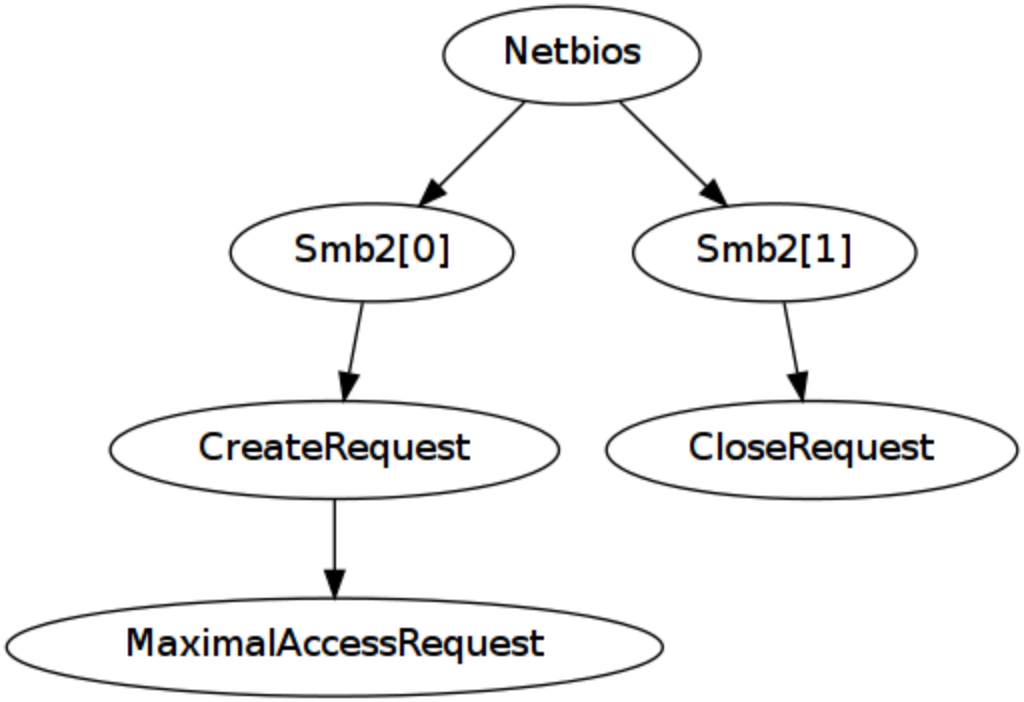
Subclassing Frame (cont.)


```
# Return list of children
def _children(self)
    return self._bars

# Encode to cursor
def _encode(self, cur):
    cur.encode_uint32le(self.some_field)
    cur.encode_uint16le(len(self._bars))
    for bar in self.bars:
        bar.encode(cur)
```

Subclassing Frame (cont.)


```
# Decode from cursor
def _decode(self, cur):
    self.some_field = cur.decode_uint32le()
    for _ in xrange(cur.decode_uint16le()):
        # Bar constructor calls parent.append(self)
        Bar(self).decode(cur)

# Helper for child frames
def append(self, child):
    self._bars.append(child)
```

Real Example: SMB2 compound



SMB2/3 Protocol (pike.smb2)

- Smb2 (header)
- Request/Response
  - Abstract superclass of command frames
  - Subclasses automatically registered in lookup table by command/structure size
- Create{Request,Response}Context
  - Same deal, but for create contexts
  - Similar pattern for ioctls, info levels, …

SMB2/3 Protocol (cont.)

- Concrete frame types
  - CreateRequest
  - OplockBreakResponse
  - LeaseRequestContext
  - …
- Various and sundry enums
  - Status
  - CreateOptions
  - …

Examples


```
class CloseRequest(Request):
    command_id = SMB2_CLOSE
    structure_size = 24

    def __init__(self, parent):
        Request.__init__(self, parent)
        self.flags = 0
        self.reserved = 0
        self.file_id = None

    def _encode(self, cur):
        cur.encode_uint16le(self.flags)
        cur.encode_uint32le(self.reserved)
        cur.encode_uint64le(self.file_id[0])
        cur.encode_uint64le(self.file_id[1])
```

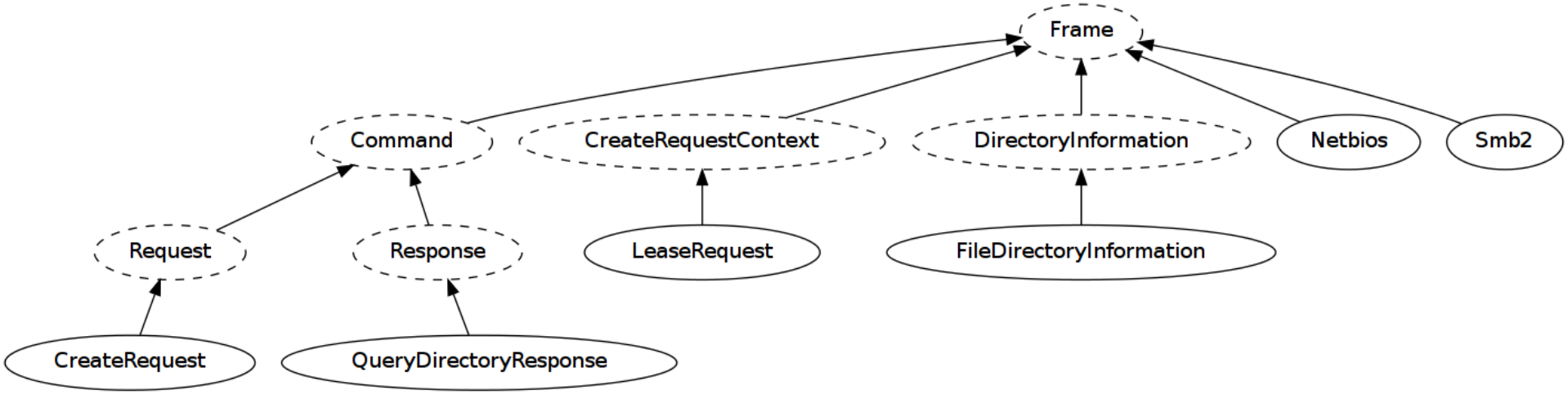
Examples


```
class MaximalAccessResponse(CreateResponseContext):
    name = 'MxAc'

    def __init__(self, parent):
        CreateResponseContext.__init__(self, parent)
        self.query_status = 0
        self.maximal_access = 0

    def _decode(self, cur):
        self.query_status = Status(cur.decode_uint32le())
        self.maximal_access = Access(cur.decode_uint32le())
```

Examples


```
# File attributes
class FileAttributes(core.FlagEnum):
    FILE_ATTRIBUTE_READONLY             = 0x00000001
    FILE_ATTRIBUTE_HIDDEN               = 0x00000002
    FILE_ATTRIBUTE_SYSTEM               = 0x00000004
    FILE_ATTRIBUTE_DIRECTORY            = 0x00000010
    FILE_ATTRIBUTE_ARCHIVE              = 0x00000020
    FILE_ATTRIBUTE_DEVICE               = 0x00000040
    FILE_ATTRIBUTE_NORMAL               = 0x00000080
    FILE_ATTRIBUTE_TEMPORARY            = 0x00000100
    FILE_ATTRIBUTE_SPARSE_FILE          = 0x00000200
    FILE_ATTRIBUTE_REPARSE_POINT        = 0x00000400
    FILE_ATTRIBUTE_COMPRESSED           = 0x00000800
    FILE_ATTRIBUTE_OFFLINE              = 0x00001000
    FILE_ATTRIBUTE_NOT_CONTENT_INDEXED  = 0x00002000
    FILE_ATTRIBUTE_ENCRYPTED            = 0x00004000
```

Partial Class Hierarchy

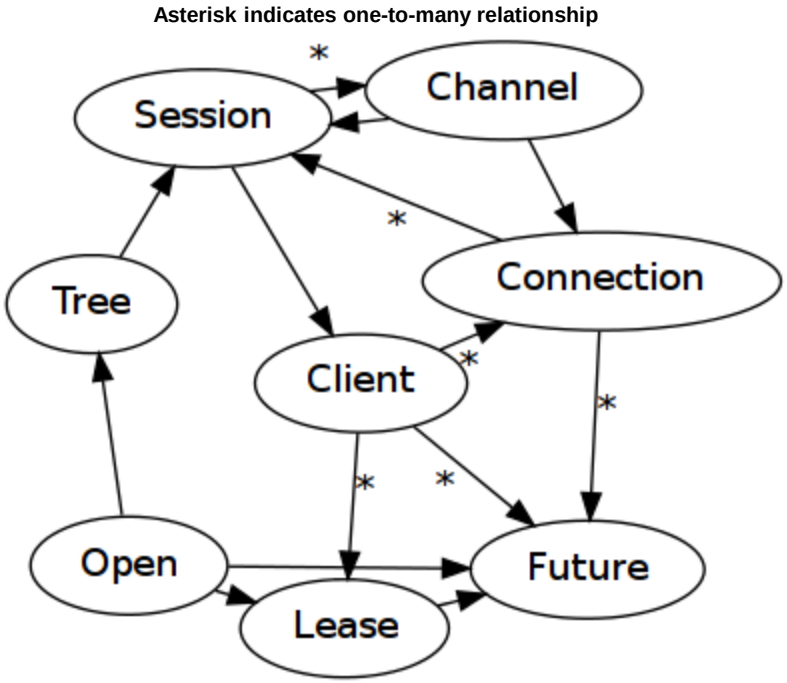


Model (pike.model)

- Implements SMB2/3 client model and functionality
- Uses Python asyncore package to allow concurrency
  - Multiple connections
  - Multi-channel
  - Async requests

Model Diagram

SMB3 Object Model + Glue



Future

- Result of asynchronous operation
- Can synchronously wait for result or nominate callback
- Rethrows exception if operation fails
- Waiting for a result runs the asyncore event loop until it is available (or a timeout occurs)
- Usually represents SMB response, but result can be anything (e.g. an Open)

Example: Using Futures


```
# Using a future synchronously
future = returns_a_future()
result = future.result()
```

```
# Using a future asynchronously
def callback(f):
    result = f.result()
    # ...

future = returns_a_future()
future.then(callback)
```

```
# Completing a future
future = Future()
future(result)
```

Client

- Represents a single client to a particular server
- Stores default negotiate parameters
  - GUID, dialects, …
- Contains one or more connections
  - This models SMB3 multichannel
- Tracks cross-connection state
  - Oplocks and leases, channel sequence, …

Connection

- Manages a single TCP connection to server
- Is an asyncore.dispatcher
- Allows requests to be submitted for sending
- Coordinates dispatching responses back to requester

Connection.submit

- The workhorse of the Pike model
- Input
  - A NetBios frame containing one more SMB2 requests
- Output
  - A list of Futures for the corresponding SMB2 responses

Connection.submit (cont.)

- Intentionally asymmetric input/output
  - Tests have strict control over chaining of requests
  - Depending on specifics of the request, responses may arrive from server in separate NetBios frames

Connection.transceive

- Input
  - A NetBios frame
- Output
  - A list of SMB2 responses
- Merely a synchronous wrapper around the asynchronous Connection.submit
- Most other interesting functions are wrappers around one of these two

Error Handling

- Connection errors (e.g. reset) cause Futures to be completed with the exception
- SMB2 Error responses handled specially
  - Future is completed with ErrorResponse instead
  - ErrorResponse inherits from Exception, so it is raised rather than returned when the Future is consumed
  - This is generally what you want

Connection Misc.

- Connection.negotiate
  - What it says on the tin
- Connection.session_setup
  - Establishes an SMB2 session
  - Creates a Session and Channel object for further use
  - Can perform SMB3 session bind when given an existing Session

Session

- Unsurprisingly, represents a session
- Stores immutable session state
  - Session ID
  - Session key from authentication
- Contains list of associated Channels
- Otherwise inert
  - Interesting operations take place on a Channel

Channel

- Represents a (connection, session) pair
- Has helper functions for sending requests after session setup
  - Channel.tree_connect
  - Channel.create
  - Channel.read
  - …
- Has helpers for building stock NetBios/SMB2 frames for further manual request construction

Channel (cont.)

- Some helpers are synchronous
  - Channel.write returns the number of bytes written
- Some helpers are asynchronous
  - Channel.create returns a Future which eventually yields an Open
- Generally, operations which could block for a long time are asynchronous
- A bit ad hoc; will be made more uniform in the future

Example: Actually Doing Something


```
chan = Client().connect(server).negotiate().session_setup()

tree = chan.tree_connect('c$')

file = chan.create(tree,
                   'write.txt',
                   access=pike.smb2.FILE_WRITE_DATA,
                   disposition=pike.smb2.FILE_SUPERSEDE,
                   options=pike.smb2.FILE_DELETE_ON_CLOSE).result()

bytes_written = chan.write(file, 0, data)

chan.close(file)
```

Tree

- Represents a tree connect
- Returned by Channel.tree_connect
- Stores immutable state such as TID and path
- Used as parameter to subsequent Channel methods such as Channel.create

Open

- Represents an open handle
- Returned (via Future) from Channel.create
- Stores a variety of state:
  - File ID
  - Oplock level or associated lease
  - Durability state
  - …
- Used as parameter to other Channel methods

Lease

- Represents a file lease
- Accessible on an Open via o.lease
- Opens with the same lease key share the same physical Lease object
- Client object maintains lease table

Using Oplocks

- Request via oplock_level parameter to Channel.Create
- Check result on an Open o via o.oplock_level
- Handle break via Open.on_oplock_break
  - Takes a function which accepts the new oplock level offered in the break notification and returns the level that the client should ack to

Example: Using Oplocks


```
handle1 = chan.create(tree, 'oplock.txt', ...,
                      oplock_level=pike.smb2.SMB2_OPLOCK_LEVEL_EXCLUSIVE)
              .result()

# Ack precisely what we are given on a break
handle1.on_oplock_break(lambda level: level)
```

Using Leases

- Request via Channel.create:
  - oplock_level = SMB2_OPLOCK_LEVEL_LEASE
  - lease_key = <guid>
  - lease_state = <state>
- Access on Open o via o.lease
- Handle break via Lease.on_break
  - Works similarly to Open.on_oplock_break

Advanced Usage

- Lower-level interfaces to handling oplock/lease breaks available
- If a helper doesn't do what you want, you can manually construct and submit exotic requests
- If a frame class doesn't do what you want, you can always subclass it and override _encode

Example: Manual Request


```
nb_req = chan.frame()
smb_req1 = chan.request(nb_req, obj=tree)
smb_req2 = chan.request(nb_req, obj=tree)
create_req = pike.smb2.CreateRequest(smb_req1)
close_req = pike.smb2.CloseRequest(smb_req2)

create_req.name = 'hello.txt'
create_req.desired_access = pike.smb2.GENERIC_READ | \
                            pike.smb2.GENERIC_WRITE
create_req.file_attributes = pike.smb2.FILE_ATTRIBUTE_NORMAL
create_req.create_disposition = pike.smb2.FILE_OPEN_IF

max_req = pike.smb2.MaximalAccessRequest(create_req)

close_req.file_id = pike.smb2.RELATED_FID
smb_req2.flags |= pike.smb2.SMB2_FLAGS_RELATED_OPERATIONS

responses = chan.connection.transceive(nb_req)
```

Test Harness (pike.test)

- Subclass of unittest.TestCase
- Quickly establish connection, session, and tree connection to server
- Host, user creds, share parameters taken from environment
- Decorators to skip tests when the server does not advertise necessary capabilities or dialect

Odds and Ends

- NT time class (pike.nttime)
- Signing and key deriviation helpers (pike.digest)

Future Work

- Increase breadth of SMB2/3 support
- Security descriptors
- Improvement to model
- NTLM story
- API documentation
- More tests!
- Patches welcome

Questions

http://github.com/emc-isilon/pike