Lecture 32 - How do Layer 2 Solutions Work

If you want to build real world applications the 3 big factors that limit what you can do are:

1. Cost of Gas on the Eth Main Chian ($30 to do a simple contract call)
2. Speed - Latency of the main chain - it is high - 22 seconds.
3. Throughput of main gain - it is low.

The thing that you want is:

1. It runs and works EVM (Solidity for example) contracts.
2. It gives you financial transactions using real Eth

For just payments there are solutions like the Lightning Network for Bitcoin. Specifically, the Lightning Network is based on state channels, which are basically attached channels that perform blockchain operations and report them to the main chain. State channels are payment channels. It not really a “smart contract” solution.

Layer 2 solutions address these problems (well... some of them better than others)...

There are 2 primary methods to the Layer 2 systems.

Zero-Knowledge Rollups

Zero-knowledge rollups – used in ZK-Rollups – are sets of data that are collateralized by a smart contract (ERC 20 type) on the main chain. The work is transported off-chain for processing. It takes about a minute to produce a block - so latency is still a problem. However it runs total at over 2,000 TPS. Also there is a tremendous level of privacy. The off chain data storage helps.

Optimistic Rollups

Optimistic rollups run on Ethereum’s base layer while moving the contract work and storage off chain. Consensus is via computation of merkle roots that have to mach between different side chains and the main chain. Optimistic rollups have to rely on a set of external validators to check the merkle roots. Dragon chain works this way.

An Example Polygon (Matic)

Polygon’s main chain is a Proof-of-Stake (PoS) sidechain. MATIC is the native token for the Polygon network. Is used as staking token to validate transactions and vote on network upgrades. MATIC is also used to pay for the gas fees on Polygon. Essentially they have an Ethereum network that is much faster that you run as a side chain.

What is involved in an optimistic rollup system

Let’s suppose that you have a staking contract - ERC777 based.

```
1: // SPDX-License-Identifier: MIT
2: pragma solidity >=0.6.0 <=0.9.0;
3:
4: import "@openzeppelin/contracts/token/ERC777/ERC777.sol";
5:
```

```
 6: contract InsLogEvent is ERC777 {
 7:
 8:     address payable owner;
 9:
10:     event AnEvent ( address indexed account, string msg );
11:     event EthSentEvent ( address indexed account, uint256 amt );
12:
13:     constructor () ERC777("Layer2Sidechain", "L2S", new address[](0)) {
14:         owner = payable(msg.sender);
15:         _mint(msg.sender, 10000 * 10 ** 18, "", "", true);
16:     }
17:
18:     modifier onlyOwner() {
19:         require( msg.sender == owner, "Sender not authorized.");
20:         // Do not forget the "_;"! It will be replaced by the actual function
21:         // body when the modifier is used.
22:         _;
23:     }
24:
25:     // Make `_newOwner` the new owner of this contract.
26:     function changeOwner(address payable _newOwner) public onlyOwner() {
27:         owner = _newOwner;
28:     }
29:
30:     function IndexedEvent ( address _acct, string memory _msg ) public returns ( bool ) {
31:         emit AnEvent ( _acct, _msg );
32:         return (true);
33:     }
34:
35:     function SendEth ( address payable _acct, uint256 _amount ) public onlyOwner {
36:         _acct.transfer(_amount);
37:         emit EthSendEvent ( _acct, _amount );
38:     }
39:
40:     // This function is called for all messages sent to
41:     // this contract, except plain Ether transfers
42:     // (there is no other function except the receive function).
43:     // Any call with non-empty calldata to this contract will execute
44:     // the fallback function (even if Ether is sent along with the call).
45:     fallback() external payable {}
46:
47:     // This function is called for plain Ether transfers, i.e.
48:     // for every call with empty calldata.
49:     receive() external payable {}
50:
51:     function withdraw( uint256 _amount ) public onlyOwner() {
52:         owner.transfer(_amount);
53:     }
54:
55:     // destroy the contract and reclaim the leftover funds.
56:     function kill() public onlyOwner() {
57:         //    Calling selfdestruct(address) sends all of the contract's current balance to address.
58:         selfdestruct(payable(msg.sender));
59:     }
60: }
61:
```

This contract has some extra stuff...

```
function IndexedEvent ( address _acct, string memory _msg ) public returns ( bool ) {
```

```
function SendEth ( address payable _acct, uint256 _amount ) public onlyOwner {
```

Now we build a side chain based on this.

First let's use postgresql for the side chain. It is fast, has tables and indexes and triggers and all sorts of stuff...

```
 1:
 2: CREATE OR REPLACE FUNCTION send_notify_after_trig()
 3: RETURNS TRIGGER AS $body$
 4: declare
 5:     msg text;
 6: begin
 7:
 8:     -- msg = '{"id":'||to_json(new.id)||'}';        -- xyzzy - use function to get full row
 9:     msg = row_to_json(NEW);
10:     PERFORM pg_notify('events',msg);
11:
12:     return null;
13: end;
14: $body$ LANGUAGE plpgsql;
15:
16: CREATE TABLE bk_block (
17:     id             uuid not null,
18:     seq            searial not null,
19:     data        text not null,
20:     hash        text,
21:     prev_hash    uuid,
22:     prev_id        uuid,
23:     whenmod        timestamp default current_timestamp not null
24: );
25:
26: create index send_notify_p1 on bk_block ( whenmod );
27:
28: CREATE TRIGGER send_notify_after_tg
29: AFTER INSERT ON bk_block
30:     FOR EACH ROW EXECUTE PROCEDURE send_notify_after_trig();
31:
32:
33:
34:
35:
36:
37:
38:
39:
40:
41:
42:
43:
44:
45:
46:
47:
48: create table "c_l2_accts_hist" (
49:     "id"                    uuid DEFAULT uuid_generate_v4() not null primary key,
50:     "acct_no"                 text not null ,
51:     "tx_from_acct"             text ,
52:     "tx_to_acct"             text ,
53:     "delta"                 money default 0 not null ,
54:     "user_id"                 uuid not null ,
55:     "seq"                    bigserial not null,
56:     "created"                 timestamp default current_timestamp not null,
```

```
57: );
58:
59: create index "c_l2_accts_hist_u1" on "c_l2_accts_hist" ( "acct_no", "seq" );
60:
61:
62: create table "c_l2_accts" (
63:     "id"                    uuid DEFAULT uuid_generate_v4() not null primary key,
64:     "acct_no"                 text not null ,
65:     "signature"             text ,
66:     "tx_from_acct"             text ,
67:     "tx_to_acct"             text ,
68:     "balance"                 money default 0 not null ,
69:     "user_id"                 uuid not null ,
70:     "created"                 timestamp default current_timestamp not null,
71: );
72:
73: create index "c_l2_accts_u1" on "c_l2_accts" ( "acct_no" );
74:
75:
76:
77:
78: CREATE OR REPLACE FUNCTION notify_for_l2_accts()
79: RETURNS TRIGGER AS $$
80: declare
81:     msg text;
82: begin
83:     if NEW.tx_to_acct is not null then        -- if we are sending funds.
84:         if not validateSignature ( NEW.signature ) then
85:             RAISE EXCEPTION 'invalid signature time:% acct:', now(), NEW.acct_no;
86:         end if;
87:         if NEW.balance < 0 then
88:             RAISE EXCEPTION 'insufficient funds time:% acct:', now(), NEW.acct_no;
89:         end if;
90:     end if;
91:     insert into c_l2_accts_hist ( acct_no, delta, user_id, tx_from_acct, tx_to_acct )
92:         values ( NEW.acct_no, NEW.balance-OLD.balance, NEW.user_id, NEW.tx_from_acct, NEW.tx_to_acct );
93:     insert into bk_block ( id, data ) values ( NEW.id, row_to_json(NEW) );
94:     return NEW;
95: end;
96: $$ LANGUAGE plpgsql;
97:
98: CREATE TRIGGER imutable_l2_accts_imutable
99: AFTER INSERT ON c_l2_accts
100:     FOR EACH ROW EXECUTE PROCEDURE notify_for_l2_accts();
101:
102:
103:
104:
105: CREATE OR REPLACE FUNCTION imutable_data()
106: RETURNS TRIGGER AS $$
107: begin
108:     return NULL;
109: end;
110: $$ LANGUAGE plpgsql;
111:
112:
113: CREATE TRIGGER notify_l2_accts_imutable
114: AFTER DELETE or UPDATE ON c_l2_accts_hist
115:     FOR EACH ROW EXECUTE PROCEDURE imutable_data();
116:
117: CREATE TRIGGER send_notify_imutable
118: AFTER DELETE or UPDATE ON send_notity
119:     FOR EACH ROW EXECUTE PROCEDURE imutable_data();
120:
```

This relies on sending messages from PostgreSQL to the contract. PostgreSQL has a nice facility to do pub/sub. So let's use that in go.

```
 1: package main
 2:
 3: import (
 4:     "database/sql"
 5:     "encoding/json"
 6:     "flag"
 7:     "fmt"
 8:     "os"
 9:     "time"
10:
11:     "github.com/ethereum/go-ethereum/accounts/keystore"
12:     "github.com/ethereum/go-ethereum/ethclient"
13:     "github.com/lib/pq"
14:     "gitlab.com/pschlump/ReadConfig"
15: )
16:
17: type PgData struct {
18:     Cmd  string `json:"Cmd"`
19:     Hash string `json:"Hash"`
20:     To   string `json:"To"`
21:     Amt  string `json:"Amt"`
22: }
23:
24: type EthAccount struct {
25:     Address         string
26:     KeyFile         string
27:     KeyFilePassword string
28: }
29:
30: type ConfigType struct {
31:     URL          string            `json:"URL" default:"http://127.0.0.1:8545/"` // Address of server, http://l
32:     ContractAddr map[string]string `json:"ContractAddr"`
33:     Account      EthAccount
34:     LogAddress   string `json:"LogAddress"`
35:
36:     // Global Data
37:     Client     *ethclient.Client
38:     LogIt      *InsLogEventControl
39:     AccountKey *keystore.Key
40: }
41:
42: var Version = flag.Bool("version", false, "Report version of code and exit")
43: var Cfg = flag.String("cfg", "cfg.json", "config file for this call")
44:
45: var GitCommit string
46: var DbOn map[string]bool
47:
48: func init() {
49:     DbOn = make(map[string]bool)
50:     GitCommit = "Unknown"
51: }
52:
53: var gCfg ConfigType
54:
55: func main() {
56:     flag.Parse() // Parse CLI arguments to this, --cfg <name>.json
57:
58:     fns := flag.Args()
59:     if len(fns) != 0 {
60:         fmt.Printf("Extra arguments are not supported [%s]\n", fns)
61:         os.Exit(1)
```

```
62:     }
63:
64:     if *Version {
65:         fmt.Printf("Version (Git Commit): %s\n", GitCommit)
66:         os.Exit(0)
67:     }
68:
69:     if Cfg == nil {
70:         fmt.Printf("--cfg is a required parameter\n")
71:         os.Exit(1)
72:     }
73:
74:     // -------------------------------------------------------------------------
75:     // Read in Configuration
76:     // -------------------------------------------------------------------------
77:     err := ReadConfig.ReadFile(*Cfg, &gCfg)
78:     if err != nil {
79:         fmt.Fprintf(os.Stderr, "Unable to read confguration: %s error %s\n", *Cfg, err)
80:         os.Exit(1)
81:     }
82:
83:     // Connect to PostgreSQL
84:     pgConnectString := os.Getenv("POSTGRES_CONN") // TODO - Change to use your connetion
85:
86:     _, err = sql.Open("postgres", pgConnectString)
87:     if err != nil {
88:         fmt.Fprintf(os.Stderr, "Unable to connect: %s %s\n", pgConnectString, err)
89:         os.Exit(1)
90:     }
91:
92:     // Main Processing
93:     logError := func(ev pq.ListenerEventType, err error) {
94:         if err != nil {
95:             fmt.Fprintf(os.Stderr, "Error on event listner:, %s\n", err)
96:         }
97:     }
98:
99:     err = Connect(&gCfg)
100:    if err != nil {
101:        fmt.Printf("Unable to connect to %s: error:%s\n", gCfg.URL, err)
102:        os.Exit(1)
103:    }
104:
105:    // Setup Contract Call
106:    ethLogEvent, err := NewInsLogEvent(&gCfg)
107:    if err != nil {
108:        fmt.Printf("Unable to instanciate the contract: %s\n", err)
109:        os.Exit(1)
110:    }
111:
112:    listenForEventToOccure := func(l *pq.Listener) {
113:        nth := 0
114:        period := 50
115:        for {
116:            select {
117:            case n := <-l.Notify:
118:                nth++
119:                if debug_001 {
120:                    fmt.Printf("Received event on chanel=->%v<- data=->%s<-\n", n.Channel, n.Extra)
121:                }
122:                var pd PgData
123:                err := json.Unmarshal([]byte(n.Extra), &pd)
124:                if err != nil {
125:                    fmt.Printf("err %s data ->%s<-\n", err, pd)
126:                }
127:                switch pd.Cmd {
```

```
128:                 case "LogEvent":
129:                     tx, err := ethLogEvent.IndexedEvent(gCfg.LogAddress, pd.Hash)
130:                     if err != nil {
131:                         fmt.Printf("err %s data ->%s<-\n", err, pd.Hash)
132:                     } else {
133:                         fmt.Printf("Success Tx: %s\n", JsonToString(tx))
134:                     }
135:                 case "SendEth":
136:                     tx, err := ethLogEvent.SendEth(pd.To, pd.Amt)
137:                     if err != nil {
138:                         fmt.Printf("err %s data ->%s<-\n", err, pd.Hash)
139:                     } else {
140:                         fmt.Printf("Success Tx: %s\n", JsonToString(tx))
141:                     }
142:                 }
143:                 fmt.Printf("%+v\n", pd)
144:             case <-time.After(60 * time.Second):
145:                 nth++
146:                 fmt.Printf("No data received after 1 min, verifying connection to database\n")
147:                 go func() {
148:                     l.Ping()
149:                 }()
150:             }
151:         }
152:     }
153:
154:     pgConn := pq.NewListener(pgConnectString, 1*time.Second, time.Minute, logError)
155:     err = pgConn.Listen("events")
156:     if err != nil {
157:         fmt.Fprintf(os.Stderr, "Unable to setup listener: %s\n", err)
158:         os.Exit(1)
159:     }
160:
161:     fmt.Printf("Event Listener Started \n")
162:     for {
163:         listenForEventToOccure(pgConn)
164:     }
165: }
166:
167: func JsonToString(m interface{}) string {
168:     s, err := json.MarshalIndent(m, "", "\t")
169:     if err != nil {
170:         fmt.Fprintf(os.Stderr, "Error: ", err)
171:         return ""
172:     }
173:     return string(s)
174: }
175:
176: var debug_001 = false
```