Distributed Smart Contracts

Mark Lester & Ben Kahn
February 2017

Contents

Proposal:

> *To implement an open standard, and underlying software layer, supporting private and indisputable sharing and editing of mutually accessible information sets across independent corporate computer systems.*
>
> *To extend this to support codeless smart document capabilities including access control to specific regions within the contract, variable constraints, derived fields, and event triggers.*

Background

Aims

We set out to address a sharing problem which occurs in commercial insurance and reinsurance, creating frictional costs which account for 25% to 30% of the premium for some products. Cost & delay arise from the need for counterparties to share, co-authenticate and agree complex, bespoke document and data sets.

Our analysis & problem cases are from insurance, but the issues described arise in many markets where non-trusting parties must share sensitive or commercial information securely. Most of what follows applies equally to security post-trade processing, government services, supply chain tracking/coordination, health records sharing and a range of other sectors and scenarios.

Hubs

Insurance is a world of byzantine business contracts, whose details are shared and co-approved by many organisations - regional and local brokers, insurers, reinsurers, syndicates, claims handlers, agencies, panels, adjusters, underwriters, regulatory authorities and other intermediaries, not to mention the insured party.

It is hard to standardise because it is not one homogeneous market. There are many party types, companies may fill hybrid or compound roles, and there is elision and overlap between categories.

The commonest solution to record sharing in a messy marketplace is a hub, a kind of MakeMyContract.com

Everyone must buy into this, including competitors and users of other hubs, of which there are many. Even ignoring resilience, scalability and maintenance, a problem of trust often impacts any centralized solution. Where use is not mandatory, there is the question of who runs the system, and do we trust them?

Companies must stick their entire business ledger on someone else's computer, not just the data which they may share with the owner of any specific remote system. This may deter them from doing it at all. If it does not, they do not regard it as the true record, and frequently internally duplicate its data and processing, to be certain they can pull out a full audit for operational and regulatory purposes.

This inability to fully trust hubs mean they often achieve only a fraction of what could be possible.

Trust

Trust is of different forms.

Do I trust your computer system
- … to tell the truth (and nothing but the truth), not get hacked/break/go off-line?
- … not to leak critical data to everyone else on the system or even the planet?
- … to behave how we agreed it would?

The first question relates to ***confidence*** in the reliability of the system.
This second one we can call ***privacy***.
The third is about ***transparency*** and relates to the computing itself…

A key benefit of blockchain is the creation of evidentially provable audit. This enables transactions between parties who otherwise would have insufficient mutual transparency to fully rely on each other's systems. You don't need to trust the other party to do the right thing if you can see real-time evidence they are.

But if rules that constrain your contract are ***coded***, now you need to trust that the code is free from bugs, intentional or unintentional backdoors & other vulnerabilities. Maybe you fully understand online security risks end-to-end. If not - and you rely on your techies - then you need to trust *them* to identify any risk.

Approval

Proof of agreement must represent business, not just IT, approval. *Meaningful* proof of agreement must represent *informed* business approval.

Concordata has a codeless interface – data structure and rules can be defined & validated by non-technical users with dropdowns & simple expressions - for exactly this reason.

This imposes upper limits to the complexity of logic that can practically be used. Other distributed ledger platforms support a range of languages to implement complex logic, but we have deliberately avoided this route. We believe code on the blockchain shifts, rather than solves, the core trust/transparency problem.

But with codeless, transparent smart contracts, there is no need to trust either counterparty or coder.

Concordata

We set out last year to deliver such a system. Most of what follows has now been implemented.

Implementation

Unanimity

An easy way to indisputably ensure N parties to a contract all hold the same version of things is to multiplex all operations (create, update and delete) on a contract between all parties, including for each 'Request', a 'Dialogue' for each party and their response to the Request. If all interactions use SSL and are signed by the transmitting party, evidence of consensus is irrefutable.

So the system gives irrefutable proof that data held in every remote party's database is actually identical.

Concordata's consensus strategy is simple. Everyone has power of veto, all counterparties must agree, and without indisputable signed verification, an update isn't accepted.

Data Model

The document tree consists of ***Elements***: either nodes (blocks or sub-trees) or leaves (fields with proposed values). They are described by a schema of 8 tables supporting Consensus and Contract concepts.

Elements have a universal identifier (***UID***) concocted from the native record ID on the generating system and that party name/address. As this UID is a potential information leak, it is obfuscated so you can't work out the real record number, and hence how much business the other side has done since you last checked.

Elements are replicated individually using the total consensus strategy described. Every party confirms with every other party that this transaction is correct/valid and the resulting effect on the record produces the given entire record hash. All transactions are signed and transmitted via SSL.

Outside the generic model, 3 further tables are required. ***Contract***: the record instance table; ***Party:*** the list of all registered parties and their public keys, and the intersection table of Party and Contract, i.e. the list of parties on a given contract – in insurance parlance, this is the "market" for the Risk.

Two other tables, ***Request*** and ***Dialogue***, handle all operations on all other tables. These are what the Transmitter and Receiver programs (see below) deal with when processing requests.

Interfaces

Request and Dialogue are the core of how replication works and may only be written to by the Concordata system. They are initially populated by hooks applied in the Object Relational Mapper interface (Sequelize) to trigger on create, update and delete operations on entities you do have access to, by default. The web server REST interface handler is just writing to a plain DB interface and has no knowledge of 3TC.

The ORM interface allows consensus-aware clients to read Request and Dialogue tables, but is abstracted such that create, update & delete operations on those entities are hidden.

Using a Node.js server level API, programmers can interface to local Concordata database with this ORM, add/edit/delete contracts & contents with trivial code and no knowledge of how all the magic happens.

3TC

Concordata supports three levels of consensus – we call it 3TC. You can't start labels in computer code with a numeral so the code itself reads ThreeTC. The three tiers are *Transport*, *Element* and *Record* level.

TIER 1: TRANSPORT

1st level of consensus is the transmission, reception & universal acknowledgement of Requests

It results in full, indisputable audit of the record for all parties to see. It is cryptographically provable how, when and by whom a record was changed.

The total number of messages and responses needed to authenticate a Request is approximately $N+\log_2 N$ for the algorithm used by Concordata (see Faster Consensus). Maximum number of messages broadcast by any single system is roughly $\log_2(N)+1$. Scalability considerations arise with party counts in the 1000s: when a record lock is needed - unavoidable for some operations - you must wait for 1000 systems to report back. Using 3TC with resilient nodes, the maximum number of messages any system must send could be as low as about 10 for a contract size of 1000, with the last few carrying almost a thousand items of proof.

We envisage contract sizes of fewer than a hundred parties, which the architecture can handle easily.

New Parties

Nuances to the protocol mainly arise in the dynamic nature of the contract specific network. We need to be able to add parties to a contract, and we need to multiplex every request to every party on the contract.

Should one party add a party while another simultaneously adds a field, we must ensure there can never be a loophole or data loss. Currently any current party can invite a new party, more development is needed to provide additional controls in this area.

Concordata allows new nodes to be added without needing to lock the record and thus not requiring all systems to respond and acknowledge the lock before proceeding.

Conflicts

Another challenging area is the dynamic structure of the record, in that new blocks, leaf nodes or fields can be added by various parties. Where ownership of sub-trees within the structure can be granted to one party only, so only they can add or remove fields at that point, we can avoid the need to lock the record. If more than one party need to change the same part of the structure, we're going to have to lock.

Assignment values to leaves are held as ***proposals*** by named parties, so there can (in principle), be more than one proposal to a specific field. Proposing leaf/field values for

verification and agreement can be asynchronous as parties only write to their part of the record – that is: make a proposal in their name.

Conflicts are possible for simultaneous operations A & B triggered by different parties – it is possible that A is executed first on some machines, and B on others B. But these operations are associative (i.e. A+B=B+A) so once the current activity spurt has subsided record hashes will match up.

Theoretically activity could be persistent and synchronized such that two transactions are always being validated at once. In this scenario, we won't settle on a hash. We don't view this as a practical concern. It would take two parties to co-operate like this, and we are not talking about anonymous entities but companies identified with Verisign, Dun and Bradstreet or similar. In any case, all they can do is lock, or rather prevent anyone else from locking, their own record.

Voting

Each proposal has a Boolean vote per party. This extra tier of consensus is often redundant as it is typically enough to know a proposal was auto-accepted by all systems i.e. each confirmed the proposal complies with pre-agreed validation rules. After all, the aim will usually be to automate as much as possible.

Some steps will require a human input. Rather than force the app designer to build a consensus field themselves, an explicit one is supplied by default and automatically set. You get a set of ticks for any proposal. You have to set a rule to enable manual validation and allow the user to validate.

Tier 2: ELEMENT

The 2nd level of consensus when we have a full set of signed YES votes for every contract Element

Persistence

The architecture consists of 3 daemons. A local web server within the firewall pointing inwards and Transmitter and Receiver with https access to all registered parties.

The Transmitter wakes up regularly, or when notified by the web server. It polls the request table for unfinished work and broadcasts transactions in strict order per party.

The Receiver handles incoming Requests. On completing a job, the Receiver notifies the web server which in turn notifies any attached client via websockets using a subject format identical to the Epilogue REST API.

The entire stack is ***reactive***; entities are updated in the client automatically via websockets without a subsequent REST call. Typically, the Receiver will respond to the remote Transmitter with completion of the job and not need to then send a message via its own Transmitter back to the originating party. The local Transmitter then notifies its local web server of database changes and it in turn notifies any listening clients.

Rules

The Generic Element Model has been extended to support a simple but powerful instruction set of rules. Rules can be applied to fields to enforce constraints and access/write control. We also support assignments whereby you can set a Rule that a

certain field is a function of one or more other fields, and that this assignment is not evaluated until the input fields have been assigned.

A hierarchical dot syntax lets you reference fields throughout the document tree with _ used as the root. For example, *_.A.B* means field B in block A at the root level, *B* just means field B in the current block (of the field you are assigning). You can also activate a block upon a trigger on a given event/value setting, and thus create a workflow. e.g. you entered such a country, there's a special tax form and some other stuff we have to do now for them.

Rules are owned by specific parties but agreed via an explicit Boolean vote. This is similar to the Proposal system and removes any need to lock the record while editing a rule. Rules always need explicit approval from all the other member parties, and don't have proposal values but a list of attributes or arguments.

Publication

Rules are inactive during editing to ensure consistency across the group. This introduces a need to publish rules simultaneously as a group, by locking the record and universally executing a flag setting on these rules.

Use cases we've analysed require some migration or workflow mode approach, typically an edit phase during which the document and rules are defined and agreed. Then the contract is "published" at which point rules are fixed, or locked down. This inevitably requires the record to be locked.

TIER 3: RECORD

The 3rd level of consensus is network wide record level locking

Locking

The *Dialogue* entity contains the hash calculated by the remote system, returned by the remote Receiver or delivered by the Transmitter. If multiple actions are being performed in parallel, hashes may not always match for a specific Request. Once the activity spurt concludes the final hashes will match and we obtain record level consensus - without the need to force a lock by ensuring all remotes have ceased processing.

A consensus aware client application can highlight on its UI that the update is still synching and a consistent set of hashes is not yet available from all parties for the last response. If it fails to synchronize after a certain period, we may wish to send an alert to the administrators, to look into matters off-channel.

The lock strategy works slightly differently to the other Request distribution. Asynchronous execution can occur as soon as a Party has proof from the originating Party. With a lock, you need to wait till you have it from everyone, then execute the Request.

There is currently no strategy to handle lock contention. Both locks fail when each Party receives the second lock request while one is still pending; an error percolates up to the client to say who you collided with. For user-driven events (i.e. when user action causes a lock to be issued) this is acceptable. For programmatical batch operations requiring locks to be obtained, there will need to be a backing off strategy.

Workflow

The “edit” phase can be simple where an agreed template contract can be imported. Once the set of access and other rules are published, the document/contract can be filled in/executed in a “live” phase.

Contingent workflow phases can be established where sets of rules are activated or deactivated upon satisfaction of a given condition. These are supported by concepts of ***Rule Sets*** and ***Triggers***.

Code

3TC is written entirely in JavaScript, and employs Sequelize, Express, Backbone, Mocha, Epilogue and many other great Node modules. The Receiver daemon is an Express instance, reading REST calls issued by the Transmitter. The Transmitter and Receiver are connected via websockets to the internally facing web server called the Concordata server, which runs the client application.

Gemini

The codebase employs a complete web stack we call ***Gemini***, which binds Sequelize and Backbone from host to client respectively. Gemini removes client side data configuration entirely by allowing a Backbone collection schema to be instantiated directly from a set of Sequelize definition files.

This is done by dumping the entire Sequelize model definition structure, after cleanup and minor transform, at the root of the API and treating it as a collection of model definitions. You define your entity definition, including any relations, on the server in a file in a models directory. The collection model is automatically defined in the client, so long as you don't contract out of this in the model definition.

A primary parent entity is defined as being the first of any *belongsTo* relations. Collections to any orphan entities are instantiated and populated up to the cache limit, and the first model instance selected. All child entities having this as their primary parent are fetched into respective collections, using the relevant parent key value, the first in the collection by default being selected and relevant child entities fetched and so on.

Reactive

Gemini is a ***reactive*** stack. When a model changes in the DB via Sequelize ORM interface, the DOM of any attached client updates automatically. Add data on a system and updates to subscribing remote clients are event-driven prompt. When you edit your screen the other guy sees it happen as soon as you hit return. This is all achieved using standard frameworks, but it is very cool all the same when you plug it all together.

Gemini allows models to be extended with configuration data usable in the client. If you create a bespoke data type "colour", the client can now embed a colour picker given that knowledge. It also exposes DB-level constraints and datatypes directly to the client so it can react on them. Backbone Views used are derived from bases that support constraint handling and enhanced data typing.

Testing

A scenario engine built upon Mocha runs both on the web UI of the prototype client application, and using the API. A series of non-trivial scenarios are available to test the system at a high level. We have found this to be a highly effective demo tool as it can act as a scenario-runner, automatically driving the UI.

This currently allows a hands-free demo of a multi-party scenario where one node calls the shots. More development is needed to extend this to support synchronised network testing and demonstration i.e. coordinate actions initiated independently by clients, but which form part of a single scenario.

Next Steps

All the above is already in existence if short on air miles. There has been significant market validation but testing, for example, has been necessarily limited by the unfunded, bootstrapping nature of our effort.

This section outlines some obvious next steps relating to

- Party discovery
- Integration to naïve, or consensus-unaware, systems
- Extensions to the API
- Merkle hashing for tamper proofing
- Improved security
- More efficient consensus strategy

Discovery

The Party table would benefit from a discovery mechanism and integration with network configuration enabling your system to accept connections from named parties. Keys are those filed by the organisation on Verisign or similar identity provider. Currently security config must be handled manually by your sysop - you tell the admins who you want to do business with, and they set up IPs, firewall rules, SSL certificates etc.

Naïve Systems

Concordata supports 3 levels of consensus, but existing client applications will not. They will expect to POST a value and receive a response only when it's made it into the receiving database. 3TC is asynchronous, you get acknowledged as soon as your request is made, but this does not mean it is in everyone else's DB yet.

By blocking on the arrival of verification from all remote parties, we can be certain of a universally accepted action. Furthermore, if we presume at most one proposal per field, we can adapt GET operations to return only proposed values which have been validated network wide. A consensus unaware system can now work without adaption, other than re-routing its REST calls a little.

If a remote system is down, the initial POST eventually times out. Reading our DB with our naive interface we won’t see the value we posted because wasn't validated by everyone. But the original request is still pending. Once the remote system comes back online the proposal will presumably be verified, but the client could be excused for posting the value again in the meantime.

Without locking the record, two parties could simultaneously propose a value to a field, causing any “only one proposal per field” rule we may have applied to fail and we’ll have two values. A consensus aware system can visually flag the field to alert a conflict to resolve, but a naïve system will be unable to do this.

A console is envisaged to warn and help clean up conflicts and timeouts in case of intermittent remotes.

API

The Concordata web server API is generated by Epilogue and is much as you'd expect,

```
/api/<entity>[/<id>[/child entity>]]]?[<attribute>=<value>...]
```

Hence

```
/api/Contract/123/Elements
```

returns all the Elements in the Contract of local primary index of 123.

At present, using it requires understanding of the internal GEM model. Document tree handling is fiddly: to get children of an Element, one needs first to work out the parent's local id. So for a given a UID, we fetch

```
/api/Elements?universalId=<universalId>
```

pull out the id field, then fetch

```
/api/Elements?parent_entity=<id>
```

Generating a tree structure needs a recursive assembler in your application using this interface.

An access point called FullContract returns hierarchical JSON of the document tree, plus hash of the record

```
/api/FullContract?ContractId=<id>
```

A more intuitive interface, for use by naive clients and enterprise service buses, might look something like

```
/new-api/<universalId>[/<node name>[/<node name>..]][/<field name>]
```

For a UID of *123456*, we can retrieve the Country field in an Address block within a Claimant block with

```
/new-api/123456/Claimant/Address/Country
```

This would permit the tree to be addressed hierarchically by element name, with no need for knowledge of the naming or purpose of internal tables and schemas, only of your document definition.

We can now use just the field names from the contract (not GEM entity names) so

```
/new-api/123456/Claimant
```

can return something like

```
<Address>
   . . .
   <Country>Mexico</Country>
</Address>
```

or

```
{
  “Address”:{
  . . .
  “Country”:”Mexico”
  }
}
```

and

```
/new-api/abcdef/Claimant/Address/Country
```

just

```
Mexico
```

This would simplify client application code which now no longer needs to be GEM- or consensus-aware.

If the URI terminates at a node rather than a leaf/field the JSON or XML of the tree below that point is returned. If you just give the record ID you'll get the whole record. POSTing at a node level will create a field in that block. Posting at a leaf/field will produce a proposal value.

We could extend this to cope with the consensus and rule application of the Element Model, perhaps with

```
...#[Rules|Proposals][/Votes|Attributes]
```

so

```
/new-api/abcdef/Endorsee/Address/Country#Rules
```

gives access to the rules for the Country variable.

```
/new-api/abcdef/Endorsee/Address/Country#Rules/3/Attributes
```

returns the attributes of the third rule.

Tamper-Proofing

The audit comprises all atomic actions on the contract in the Request table, validated by related Dialogues. You cannot add undetectable erroneous entries into it without possessing private keys of ALL parties, but it is theoretically possible to delete them from the head or within the history.

There is a universally agreed hash on any lock and most transactions. Hashes are consistent and identically sequenced on all systems. Binding these in a Merkle tree provides provable protection against modification, (a 4th tier or Audit Level Consensus) because all systems can see the head of each other's Merkle.

This doesn't seem a pressing task for cases we have modelled, where the live contract is something that just gets written to, nothing gets deleted. But given a life cycle where a subtree could be created, acted upon, and then deleted without trace, it is conceivable a phantom transaction could be inserted into one system, broadcast and validated everywhere, and then removed from one or more other systems.

An industry backed Merkle package on Node handles this and we have already done work to integrate it.

Validation of the entire trail for a specific contract would require the tree to be checked for consistency then the head to be compared. This leads to a tamper proof database as secure as the product of the systems on the contract. That's not as good as blockchain where every transaction has the backing of thousands of machines. But it is still the arithmetic product of every corporation on the contract. So it's as good as your own, times anything everyone else can bring.

Security

An API lets programmers easily write and edit contracts. Concordata web server simply opens a Sequelize DB connection with this interface, automatically generates REST end points to it using Gemini and Epilogue.

Core internal tables Request and Dialogue should be made private and only privileged programs given access to them. Currently your client program, as a side effect of creating a Proposal, calls trigger code which in turn directly create Request and respective Dialogue rows, so your process needs access to them.

By enhancing Transmitter to work as an Express server, take deferred REST requests triggered from the ORM API, perform the insertion and return promptly, leaving another thread to perform remote communication which may fail or at least be tardy, we can limit programs that need write access to Request and Dialogue to only Transmitter and Receiver. These can be put in a special group and handed over to DB admin to secure.

The total assurance is the now product of the very best that the IT departments can muster for all parties to a contract. Only these two internal programs Receiver and Transmitter need details of the local private key anyway so even as things stand we can keep that to a private group.

Faster Consensus

We consider it unlikely that there are compelling business cases for Smart Contracts with very large party counts. Such contracts would, in any case, face challenges securing record locks, as the chance increases of at least one system being down as the network grows.

Our current approach is to multiplex every request between every node. This is simple and works perfectly well for party counts in single figures. For a network of N nodes, this results in N^2 messages per interaction, so there is a limit to how far it will scale.

This strategy can be radically improved. If a node passes every approval it holds to any party it contacts, and receives in return all those the remote party has, we can accumulate consensus much faster. We have an approximate method of selection of the next Party to contact, from any other Party, that ensures we always pick a Party not yet contacted by anyone until we reach saturation. In a balanced network, the maximum number of requests required by any single node should reduce to $Log_2(N) + 1$.

Roadmap

We have an arbitrary hierarchical record shared with irrefutable proof across a reactive network. Sometimes we refer to it as a *Document* to emphasize its tree nature, but *Contract* is more natural in a fintech context.

Added to this is a simple but very powerful constraints and triggering rule framework. So this is also a smart contract system which doesn't behave like the ED209 when things go wrong. It doesn't have any code, the most you do is some accounting equation in a constraint or assignment, or set up access constraints imposed on parts of the record. It's not going to force anyone to do anything.

Contracts are reusable. By sharing a Contract modelling some business construct or process, the parties you shared it with can now reuse it to transact with others. The arbitrary structure means any contract part can be included in another as a sub-contract. Contract components can be modularized and standardized also.

Evolution

Where markets must share complex data between groups, this presents an evolutionary alternative to creating consortia, and endlessly debating message and process standards.

The approach described creates a Darwinian environment of descent with modification, as contracts are shared, tweaked and optimised between market participants, and weaker offerings discarded in favour of stronger so that ultimately optimal processes emerge/evolve by selection rather than being defined by the great and the good, and reflect nuances of market transactions rather than the trade-offs of standards sets.

We believe Distributed Smart Contracts have the potential to be the WWW of business. With certificate authorities providing the core authentication of the keys, you can make an irrefutable shared contract of any form with anyone else who uses a Concordata-like platform.

Universal

With current technology, data must be held in the same number of distinct implementations, and possibly captured as many times also. Synchronisation and refinement of process would be severely restricted.

We can use standard revision control systems to arbitrate changes, the point is about a universal network to use them on. This does not require all to buy into one global system, the deal is there is no centralized hub, your data resides only on systems where it should. You pay for your own resilience, so does everyone else.

This is an approach which everyone who has a computer system can use. Smaller companies, maybe less paranoid about their weekly sales performance becoming public knowledge, can use this to deal with large companies who may soon require this kind of distributed architecture.

With multi-party commitments effortlessly shared with trust & privacy, and evolution of common standards driven by the market user base, not only can data capture costs plummet (one-touch capture for the entire supply chain) but downstream parties can

in turn provide products to support these contracts. Business opportunities that don't yet exist can arise if we have this network.

B2B, B2C, C2B

Distributed Smart Contracts seems to be a B2B play, with little B2C application. To justify its use there must be a need to selectively share, with trust/certainty and privacy, data with only those people who should have access to it. Extremely large groups are unlikely to be manageable, but with thousands of members to a contract it ain't all that secret anyway.

But there are interesting applications for allowing individuals control over data sharing with corporates – you might call this C2B. For example, it could allow arbitrarily fine control over access to individuals' health record. A platform-agnostic ability to set transparent sharing rules for health records (eg scans and results) under fixed conditions (eg time period) with specific strangers (eg consultants) may help address the web of security, privacy and consent issues which collectively deter adoption of health record sharing platforms.

Open Source

We believe anything like this must be open. Closed systems lack the peer review which ensures security, but also suffer from trust issues outlined above, which prevent companies fully relying on external platforms.

Multi-Lateral

Global corporations often comprise many operating companies delivering distinct services, for whom IT and business considerations can pull in opposing direction. The parent wished to cross-rationalise cost, but also for regulatory, operational or disposal reasons, keep sub-entities distinct – with loosely integrated systems.

In a federated set-up, Distributed Smart Contracts could help facilitate rationalisation, consolidate tracking & audit and deliver control & transparency across subsidiaries, while keeping systems wholly independent.

Many potential applications arise beyond FinTech:

- Secure sharing across Governmental departments (regtech)
- Irrefutable document sharing for the legal industry
- Remove duplicate filing and surface provenance across (many types of) supply chains
- Tracking and synchronization without subscription to a common hub

Bi-lateral

Many two-party contracts are substantial and constantly in flux. Consider providers such as car hire or travel agents who negotiate complex, bespoke tables of business rates with corporate clients. Things can vary within set rules on both sides. Providers rates may reflect peak hours or seasonal/country variations. Client workforce may move restlessly between locations, or have department-specific discounts.

These details change regularly and the service vendor needs to update them and the client to be aware and staff instructed to avoid those dates unless given approval. Bookings form amendments to the contract and are added as sub blocks. Hosting the solution entirely on either agent or client's system requires trust from the other party. A distributed solution not only solves that, but also creates an open market.

A contract encapsulates a type of transaction – data and behaviour – and a solution provider could resell this to vendors or direct clients, or help to cross-connect all these parties. Vendors can offer a choice of purchasing models, clients can choose between them.

Blockchain

The sermon above may have you thinking “isn’t this what blockchain is going to do?” Well, not exactly, no.

Blockchain was designed to solve problems related to implementing a crypto currency, such as prevention of double-spending of an e-token, and its architecture reflects this. Concordata tackles a different problem: evidencing alignment of information and processing across multiple independent corporate systems.

Blockchain is a transactional system. You issue transactions to be validated onto the chain. This is too low level for most applications to interpret. Applications expect a database record they can read and update. At a minimum, middleware is needed between this “transport” level of authentication and what that means to the associated chain/contract/record/document. If this is in some other system, then that other system must be tightly integrated and cross checkable with the chain to stand up any immutability claim.

It is challenging to obfuscate connectivity information in a blockchain in a “legitimate” context i.e. not using a Tor server or deferring trust to some other hub. You can find this discussed on blogs of lead engineers at many notable blockchain projects. Or just look at what you must do to perform a secret bitcoin transaction. It is harder still to obfuscate a program and still be able to run it. For the network as a whole to validate an operation, you must expose that operation and its inputs and expected outputs. No practical solution has yet been proposed for this, although plenty of able people have given the matter considerable thought.

Blockchain is a money thing, that’s what it does. It addresses value transfer problems, not contract sharing problems. It turns out arbitrary, mutual, private record sharing on blockchain isn’t easy and scales poorly. As yet, we have not seen an example that will not require nearly all of what is described here.

You don’t need blockchain for irrefutable proof someone said something, standard PKI encryption will do. Blockchain extends PKI with universal ordering across updates. We don’t need that either, we aren’t running a universal currency, only ordering transactions within a contract, when needed, and we have that already.

There is then the matter of just how immutable you want/need to be. If you aren't totally immutable then you basically aren't immutable. As we have already seen with some emerging smart contract technology, engraving your fate onto an immutable ledger can have unexpected and unpleasant side effects. And immutability is not free. There is a $$ cost to running a platform indefinitely, however distributed.

Proof of Work is discarded in these permissioned approaches, the rationale being that these will be entirely administered, finite networks. Great news if you are in the corporate network administering business. Not only will the network need tight control, but applications must be closely monitored to prevent flooding of a network. As with immutability, this undercuts the perception that there is no cost to blockchain validation.

As network size grows to justify, so does the risks of security, DDoS & other network disruption via malicious or bad programs. Even with the bluest-chippiest infrastructure, the cost of protecting against this will grow.

The definition of the term “*blockchain*” often depends on who you talk to, and much positioning is still up for grabs. One route permissioned solutions could take is to strip away blockchain elements except what is discussed above - up to Tier 1 anyway and the tamper proof stuff.

If that means a blockchain of single transaction blocks, on a separate chain per contract, with a common port, requiring 100% consensus, then so be it. If not, drop the term and call it a distributed ledger. Either way, we will stand by the 3TC concepts set out above.

Conclusion

A significant software layer has been developed, employing a wealth of cryptographic, MVC, ORM and other currently available technologies. It implements a simple yet incontrovertible total consensus policy.

Our approach is inclusive. Application developers can easily build and deploy solutions. The codeless nature of the system allows analysts to easily derive and extend these without IT involvement. We believe this combination has the potential to drive viral adoption.

Considerable personal commitment has gone into development in the last year, but it is no longer practical to progress without substantial collaboration. If you'd like to learn more with a view to contributing to or leveraging the codebase for your own project, please get in touch.

mc_lester@yahoo.co.uk
benjaminkahn@gmail.com