





MAARİF MEKTEPLERİ 59

© Maarif Mektepleri Eğitim Basın Yayın Danışmanlık
Org. Araş. İnş. Tur. San. ve Tic. Ltd. Şti.
Maarif Mektepleri Yayınları, Ankara

Algorand İle Blokzincir Geliştirme

Yazarlar: Buğra Ayan - Enes Polat

Yayın Yönetmeni: Raşit Akyüz

Editör: İbrahim Sertkaya

Redaksiyon: Hakan Yalçın

Tasarım ve Uygulama: GRİFFİN [0312 419 1619]

Kapak Tasarım: GRİFFİN

Birinci Baskı: Mart 2021

Baskı: Tarcan Matbaa

Sertifika No: 47663

Adres:

İvedik Cad. No: 417 Yenimahalle/ANKARA

ISBN: 978-605-06679-9-8

Sertifika No: 44141

Adres:

Kızılay Mah. Gmk Bul. Fevzi Çakmak 2 Sok.

No: 37/1 Çankaya /Ankara

0312 419 16 19

www.maarifmektepleri.com.tr

info@maarifmektepleri.com.tr

maarifmektepleri@gmail.com

facebook.com/maarifmektepleri

instagram.com/maarifmektepleri

Bu kitaptaki hiçbir bilgi yatırım tavsiyesi değildir.

ALGORAND İLE BLOKZİNCİR GELİŞTİRME

BUĞRA AYAN & ENES POLAT



BUĞRA AYAN

Buğra Ayan, 2012 KTÜ Elektrik Elektronik Mühendisliği bölümü mezunudur. 2013 yılında Milli Savunma Bakanlığı Araştırma Geliştirme Daire Başkanlığı'nda Hava ve Uzay Sistemleri'nde mühendis olarak göreve başlayan Ayan, 2015 yılı itibariyle Cumhurbaşkanlığı İdari İşler Başkanlığı'nda görev yapmaktadır. Gazi Üniversitesi Bilişim Enstitüsü'nde yüksek lisans çalışmasını tamamlayan Ayan, doktora çalışmasına Gazi Üniversitesi Bilişim Enstitüsü'nde devam etmektedir.

E-posta: bugra.ayan@yahoo.com

ENES POLAT

Enes Polat, 2008 KTÜ Elektrik Elektronik Mühendisliği bölümü mezunudur. 10 yıldır haberleşme sektöründe çeşitli görevlerde bulunmaktadır. 2017 yılından itibaren Türk Telekom Erzurum Bölge Müdürlüğünde Enerji ve Soğutma Sistemleri Müdürü olarak görev yapmaktadır. Enerji ve haberleşme alanlarında görevini sürdürmektedir. Özel ilgi olarak Yapay Zeka, IOT, Android, IOS, Flutter, Python, OpenCV, Kotlin, Linux, görüntü işleme ve Siber güvenlik alanlarıyla uğraşmaktadır.

E-posta: polat.enes1985@gmail.com

İÇİNDEKİLER

BİRİNCİ BÖLÜM

ALGORAND İLE BLOKZİNCİR GELİŞTİRME.....	7
Blockchain Temelleri.....	7
Blokzincir Teknolojisinin Tarihçesi.....	8
Blokzincir’de Genel Kavramlar.....	12
Peki, Blokzincirler Ne Yapar?.....	13
Blockchain Uygulamaları.....	13

İKİNCİ BÖLÜM

MERHABA ALGORAND.....	15
Algorand.....	15
Pure Proof-of-Stake.....	16
Blokzincir’in Üçlemesi: Güvenlik, Merkeziyetsizlik, Ölçeklenebilirlik.....	18
Algorand Üzerinde Tekli ve Çoklu İmzalı StandAlone	
Hesap Oluşturma.....	20
Algorand Cüzdanı.....	23
Algorand Ortamı Kurulumu.....	24
Algorand Mimarisi.....	24
Araç Seçimi.....	25
Yazılım Geliştirme Kitleri (SDK).....	25
Komut Satırı Arayüzü (CLI) Araçları.....	25
Dizin Oluşturucu.....	26
Ağ Seçimi.....	26
Algod Adresi ve Tokeni Elde Etme.....	27

ÜÇÜNCÜ BÖLÜM

PYTEAL İLE AKILLI KONTRATI OLUŞTURMA.....	29
Atomik Transferler.....	33
Yinelenen Takas Sözleşmesi.....	39
Paylaşımlı Ödeme.....	46
Periyodik Ödemeler.....	54
Oylama Sistemi.....	60

DÖRDÜNCÜ BÖLÜM

REACH İLE DAPP KODLAMAYA GİRİŞ.....	73
Windows'a Reach Kurulumu.....	74
Reach Üzerinde Taş Kâğıt Makas Oyunu Yapma.....	77
Reach Üzerinde Taş Kâğıt Makas Oyunu Yapma.....	81
Taş Kâğıt Makas Oyununa Puan Sistemini Ekleme.....	86
SIK SORULAN SORULAR.....	93

BİRİNCİ BÖLÜM

ALGORAND İLE BLOKZİNCİR GELİŞTİRME

Blokchain Temelleri

Gelin, sonda söyleyeceğimizi başta söyleyerek başlayalım. Blokzincir geliştirme süreci size yorucu gelebilir, hatta “Nereden çıktı bir de Block-chain?” diyebilirsiniz. Ama bir süre sonra muhtemelen “İyi ki de çıkmış!” diyeceksiniz. Tarihte cep telefonları, bilgisayarlar gibi bazı icatlar vardır. Düşünür ve deriz ki, “Biz bundan önce nasıl yaşıyormuşuz?”

Blokzinciri ve onun felsefesini anladıkça da insan benzer cümleler kurabiliyor. Mesela çok sevdiğiniz bir şarkıcının albümünü satın aldığınızı düşünün. Gerçekten verdiğiniz paranın hakkaniyetli bir şekilde sanatçıya, ses emekçilerine, albümü size nakliyesini yapan görevliye dağıldığından nasıl emin olabilirsiniz? Olamazsınız. Hatta Türkiye’nin en çok izlenen filmle-
rinde de böyle bir süreçten dolayı insanların davalı olduğunu biliyoruz.

Hayat güven üstüne kuruludur. Örneğin bu satırları okurken belki benim isimden dolayı belki de konunun kulağınıza gelme şeklinden dolayı buna bir güven atfedip vakit ayırıyorsunuz. Peki benim bu kitabı en olması gerektiği şekilde yazdığımın nasıl emin olabilirsiniz? Olamazsınız. Tersini düşünelim. Bu kitaba ücretsiz olarak sahip oldunuz. Yazar sizin konuya ilgi duyabileceğinizi ve bunun da ekosistem açısından faydalı olabileceğini düşünerek size bu kitabı gönderdi. Peki sizin kitabın yüzüne bile bakmadan başka birine satmayacağınızdan veya masa altına tahta yerine kullanmaya-
cağınızdan nasıl emin olabilirim? Olamam. Fakat size güvenebilirim, siz de bana.

Bugünün dünyasında zihnimizde dolaşan milyonlarca akıllı kontrat vardır ve bu kontratların arkaplanında kriptoloji değil güven duygusu yatar. Ama derler ya insanoglu çiğ süt emmiş, o misal bu sistem sürekli sü-
rekli çöker.

Fakat bilgisayar teknolojilerinin gelişmesiyle güven duygusunun yerini yavaş yavaş yazılımlar almıştır. Peki Blokzincir savunucular bunu yeterli görür mü? Elbette hayır. Onlar “Gelin şu tek merkeze dayalı güveni dağıtalım ve binlerce merkez oluşturalım.” derler. Çünkü bu şekilde bir merkezi

olmayan bir şifreleme kayıt defterleri oluşturabilirsek işte o zaman o kayıt defterlerine değerli varlık transferlerimiz de yazılabilir. Tabi en değerlisi olarak görülen “para” da. İcinizden bir “Bitcoin” dediğimizi duyar gibiyim.

Kripto para blokzincirin küçük bir parçası mı, yoksa pratikte kripto paralar olmasa kim tanırdu blokzinciri mi kavgası teknoloji mahfillerinde devam eder durur. Tavuk mu yumurtadan yumurta mı tavuktan bilmecesi gibidir. Fakat tartışılmayan tek şey her ikisinin de geleceğin dünyasında sadece teknik değil aynı zamanda ekonomik, politik ve sanatsal bir dönüşümün doğum sancılarını taşıdıklarıdır.

Biz biliyoruz ki internet web 2.0 evresine geçtiğinde doğan sosyal ağlar dünyadaki ekonomiyi, politikayı, sanatı, kültürü ve sayamayacağımız bir çok alanı etkiledi. İşte o etkiyi alın büyük bir sayıyla çarpın ve Web 3.0 etkisini düşünün.

Web 3.0’ın çocuklarından biri Blokzincir. Onun ekosisteminde gelişecek dağıtık sosyal medya platformları, alış veriş siteleri, akıllı kontratlar da Web 3.0’ın torunları belki. Tanımlamalara kızanlar olabilir ama yorum yapanların bir beyin fırtınası sürecinde olduğumuzu unutmadan yargılamalarını dilerim.

Hadi gelin işin tarihçesiyle konuşmaya başlayalım.

Blokzincir Teknolojisinin Tarihçesi

Birçoğumuzun Blokzincir teknolojisinden haberdar olması Bitcoin’in konuşulmaya başlamasıyla oldu. Bitcoin üzerinde ilk pratik uygulaması 2009 yılında yapılmakla birlikte Blokzincir teknolojisi elbette kripto para transferinden çok daha öte bir şey. Merkezsiz yeni bir insan ve dünya hayali. Fakat her ne olursa olsun Bitcoin’e atıfta bulunmadan Blokzincir’in doğuşundan bahsetmek haksızlık olurdu.

İlk olarak isterseniz Blokzincir teknolojilerinin gelişmesine giden süreçte bakalım. 1960’lı yıllarda bilgisayar ağları geliştiriliyor ve 1969 yılında Amerika Birleşik Devletler Savunma Bakanlığı bünyesine bağlı ARPA tarafından geliştirilen ve internetin öncüsü olan meşhur ARPANET kuruluyor. Aslında daha bu yıllarda kriptografinin ağlar üzerinde nasıl kullanılabileceğiyle ilgili çalışmalar da başlıyor. Hemen 1972 yılında Ulusal Standartlar ve Teknoloji Enstitüsü tarafından Veri Şifreleme Standartları (DES) isimli bir çalışma başlıyor ve ilerleyen yıllarda yayınlanıyor. Yani internet teknolojileri gelişirken bir yandan da kriptografinin bu ağlarda nasıl kullanılacağı sürekli olarak birilerinin gündeminde oluyor.

Tarih yaprakları 1976 yılının Kasım ayını gösterdiğini IEEE üzerinde yayınlanan bir makalede hash fonksiyonlarından bahsediliyor. Doğrudan hash kelimesi geçmese de tek yönlü bir şifrelemeden bahsediliyor.

New Directions in Cryptography

Invited Paper

WHITFIELD DIFFIE AND MARTIN E. HELLMAN, MEMBER, IEEE

Abstract—Two kinds of contemporary developments in cryptography are examined. Widening applications of teleprocessing have given rise to a need for new types of cryptographic systems, which minimize the need for secure key distribution channels and supply the equivalent of a written signature. This paper suggests ways to solve these currently open problems. It also discusses how the theories of communication and computation are beginning to provide the tools to solve cryptographic problems of long standing.

The best known cryptographic problem is that of privacy: preventing the unauthorized extraction of information from communications over an insecure channel. In order to use cryptography to insure privacy, however, it is currently necessary for the communicating parties to share a key which is known to no one else. This is done by sending the key in advance over some secure channel such as private courier or registered mail. A private conversation

Bugün yaklaşık 20.000 atıf alan bu makale ile aslında “Yeni Dağıtık Dünya”nın tohumları atılmaya başlanıyor. Aşağıda çalışmadan bir görsel görüyoruz.

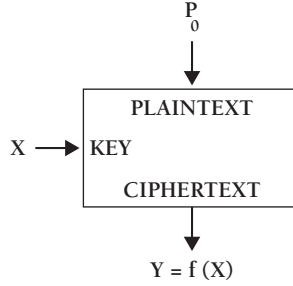


Fig. 3. Secure cryptosystem used as one-way function

Bu çalışmanın da ışığıyla 1978 yılında Martin Hellman ve arkadaşları herkese açık anahtar kullanılarak şifreleme yapabilen fonksiyonu üretti. İlginç olan bir nokta da burada ekibin ilhamını Rönesans döneminde kullanılan şifrelemelerden almasıydı.

1979 yılında Merkle Ağacı olarak sık sık duyacağımız alt alta sıralanmış bloklar gibi düşünebileceğimiz tabiri caizse hash ağacı keşfedildi.

1980 yılında TCP/IP protokolü geliştirildi ve Merkle ağacının isim babası Ralph C. Merkle kriptosistemler için protokoller geliştirilmeye başlandı.

1982 yılında Kör İmza olarak dilimize geçen, mesajı gönderen ve alan arasında imzalama işlevine sahip olan fakat mesajın içeriğini göremeyen bir katmanın olduğu Blind Signatures keşfedildi. Aynı yıl ileride detaylı değineceğimiz ve kafamızın içinde askerler gezdirecek Bizanslı Generaller Problemi bir araştırmada gündeme geldi.



The Byzantine Generals Problem

LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE
SRI International

Reliable computer systems must handle malfunctioning components that give conflicting information to different parts of the system. This situation can be expressed abstractly in terms of a group of generals of the Byzantine army camped with their troops around an enemy city. Communicating only by messenger, the generals must agree upon a common battle plan. However, one or more of them may be traitors who will try to confuse the others. The problem is to find an algorithm to ensure that the loyal generals will reach agreement. It is shown that, using only oral messages, this problem is solvable if and only if more than two-thirds of the generals are loyal; so a single traitor can confound two loyal generals. With unforgeable written messages, the problem is solvable for any number of generals and possible traitors. Applications of the solutions to reliable computer systems are then discussed.

1985 yılında eliptik eğri kriptografisi üzerine bir araştırma yayınlandı. Bu kriptografinin kullanıldığı dijital imza algoritması sonralarda Bitcoin tarafından fonların yalnızca hak sahipleri tarafından harcanmasını sağlamak için kullanılacaktı.

1991 yılında bir çalışmada Dijital Dokümanlar için nasıl bir zaman damgası kullanılabileceği üzerine araştırma yayınlandı. İşte bu araştırma Blokzincirlerin fikir babası olarak anılıyor. Yani başlangıçta bir paranın değil de bir dijital dokümanın aktarılması üzerinden yapılan araştırmalar var.

Published: January 1991

How to time-stamp a digital document

Stuart Haber & W. Scott Stornetta

Journal of Cryptology, 3, 99–111(1991) | Cite this article

5204 Accesses | 276 Citations | 664 Altmetric | Metrics

1992 yılında muhtemelen spam e-postalardan bunalan iki araştırmacı, bu e-postalarla mücadele için nasıl bir yol izlenebilir derken ileride sık sık duyacağımız Proof of Work mekanizmasını ilk kez kullanmış oldular.

ARTICLE

Pricing via Processing or Combatting Junk Mail

Authors: Cynthia Dwork, Moni Naor [Authors Info & Affiliations](#)

Publication: CRYPTO '92: Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology • August 1992 • Pages 139–147

159 0



1995 yılında, kör imza teknolojileri üzerine çalışmalar yapan David Chaum kriptografik fonksiyonları kullanarak anonim kimlikten ödeme ya-



pabilen bir elektronik ödeme sistemi geliştirdi. Digicash isimli bu sistem 1998 yılında battı. Bu da bize tekrar şunu hatırlatıyor. Bazen müthiş teknolojik fikirleriniz olsa da toplumların gerek fikri gerek teknik altyapısı bu teknolojilere hazır olmayabiliyor.

1998 yılında Nick Szabo isimli bilgisayar bilimcisi, Bitcoin'in duyurulmasından aylar önce Bitgold adını verdiği ve Bitcoin'e çok yakın bir para birimi hayalini ortaya koydu. Bu projede de Bizanslı Generaller Problemi kullanılıyordu. Nick Szabo hali hazırda akıllı kontratlar üzerine çalışmalarına devam ediyor.

1999 yılında P2P kullanan ve ağırlıklı olarak müzik dosyalarının paylaşıldığı Napster isimli dosya paylaşım uygulaması kullanılmaya başlandı. Aynı yıl Belçika'da da dijital zaman damgasının güvenlikte nasıl kullanılabileceğiyle ilgili TIMESEC isimli araştırma yayınlandı.

2000 yılında çocukluğumuzun favori programı Winamp'ın geliştiricisi Justin Frankel tarafından ilk merkezi olmayan dosya paylaşım uygulaması Gnutella geliştirildi.

Burada bir soluklanalım. Dikkat ederseniz bir yandan internet kullanıcılarının noktadan noktaya dosya transferi konusunda kullanım alışkanlıkları artıyor, bir yandan kör imza teknolojileri geliyor, bir yandan da elektronik para transferi hayalleri kurulmaya başlanıyormuş. Bunları birbirine yaklaşan parçalar gibi düşünebiliriz. Bugün bu parçaların birbirini tamamlayıp büyük bir ekonomi ortaya çıkarmasını konuşuyoruz.

2002 yılında Adam Black spam e-postalara karşı hashcash isimli bir hash algoritması geliştirdi. Bu algoritmanın geliştirilmesi 1997'lere kadar gitse de araştırma makalesi 2002 yılında yayınlandığı için bu tarihi aldık.

Hashcash - A Denial of Service Counter-Measure

Adam Back

e-mail: adam@cypherspace.org

1st August 2002

2004 yılında B-Money isimli bir anonim ve dağıtık elektronik ödeme sistemi geliştirildi. Bu ödeme sistemine Bitcoin'in icadında da atıfta bulunulacaktı.

2005 yılında, ilerleyen bölümlerde detayına değineceğimiz kriptografik puzzlelere karşı saldırıları önleyebilen bir mekanizma geliştirildi.

2009 yılında ise Bitcoin icat edilerek, onlarca kişinin sürüklediği parçalar buluştu ve dünya parayı yeniden tanımlamaya başladı.

Bu kısma bu kadar detaylı değinmemiz şu açıdan önemli. Roma bir gecede kurulmadı. Bitcoin'e giden süreçte onlarca insan harika araştırmalar yaptı ve bu sürecin parçalarını çözdü. Bugün de Algorand'ı veya diğer



blokzincir projelerini anlamaya çalışırken bu süreçlerdeki bileşenlerin aklımızda olması önemli. O zaman gerçek anlamda hayal kurabilir ve ortaya bir yenilik çıkarabiliriz.

Blokzincir'de Genel Kavramlar

Şimdi biraz da genel kavramları konuşalım ki, zihninizde bu yeni evren daha berrak bir şekilde tasavvur edebilsin.

Genel blok zincirleri: Bitcoin gibi halka açık blok zincirleri, yerel bir kripto para birimi üzerinden çalıştırılan büyük dağıtılmış ağlardır. Bir kripto para birimi, iki taraf arasında alınıp satılabilen benzersiz bir veri parçasıdır. Herkese açık blok zincirleri, herkesin herhangi bir düzeyde katılmasına açıktır ve topluluklarının koruduğu açık kaynak koduna sahiptir. Elbette bunu sadece kripto para için düşünmemize gerek yok farklı değer aktarımları için de olabilir.

İzin verilen blok zincirleri: Ripple gibi izin verilen blok zincirleri, bireylerin ağ içinde oynayabilecekleri rolleri kontrol eder. Hala büyük ve yerel bir belirteç kullanan dağıtılmış sistemlerdir. Temel kodları açık kaynak olabilir veya olmayabilir. Bu kitap yazılırken Ripple davası devam ediyordu.

Özel blok zincirleri: Dağıtılmış defter teknolojisi (DLT) olarak da bilinen özel blok zincirleri daha küçük zincirler olarak düşünülebilir. Dışarıya açık olmadığı için bu ağlarda bir token veya kripto para birimi kullanmanın da pek esprisi yoktur. Üyelerin hareketleri gözetim altındadır. Bu tür blok zincirleri, genellikle güvenilir üyeleri olan ve gizli bilgiler alıp satan konsorsiyumlar tarafından tercih edilmektedir.

Her üç tür blok zinciri de, herhangi bir ağdaki her katılımcının, kuralları uygulamak için merkezi bir otoriteye ihtiyaç duymadan defteri güvenli bir şekilde yönetmesine izin vermek için kriptografi kullanır. Tek bir merkezi otoritenin veritabanı yapısından kaldırılması, blok zincirlerinin en önemli ve güçlü yönlerinden biridir.

Blokzincir çalışırken göreceğimiz bir bilgi şudur. Blokzincirler silinmez, anonim ve güvenli yapılardır. Bu genel bilginin her koşul için istisnası vardır. Örneğin blok zincirleri kalıcı kayıtlar ve işlem geçmişi oluşturur, ancak hiçbir şey bunun sonsuza dek kalıcı olduğunu ispatlayamaz. Kaydın kalıcılığı, ağın güvenilirliğine ve sağlığına bağlıdır. Eğer bütün madenciler anlaşır ve kayıtları değiştirirse ne olacak? Hatta hepsine gerek yok sadece %51'i kayıtları değiştirirse bile akış değişebilir.

Benzer şekilde başlangıçta kimlikler anonim olarak düşünülmüştü ama bir kripto para borsasına kaydolurken neredeyse size şiir okutup kaydedecekler. Elbette bu borsalara kaydolmanıza teorikte gerek yok ama işin pratik işleyişinde bugün çoğu kullanıcının kim olduğu belli.



Güvenliğe gelirsek Ethereum'un yaşadığı siber soygunları araştırabilirsiniz. Fakat bunlar elbette istisnadır, sadece bilinmesinde de fayda vardır.

Peki, Blokzincirler Ne Yapar?

Bir blok zinciri, veri akışını yöneten merkezi bir otoriteye sahip olmayan noktalar arası bir sistemdir. Veri bütünlüğünü korurken merkezi kontrolü kaldırmanın en önemli yollarından biri, geniş bir dağıtılmış bağımsız kullanıcı ağına sahip olmaktır. Bu, ağı oluşturan bilgisayarların birden fazla yerde olduğu anlamına gelir. Bunlar bilgisayarlar genellikle tam düğümler olarak adlandırılır. Bir çok Blokzincir teknolojisini geliştirirken de bu tam düğümlerin kurulumuyla karşılaşırız.

Ağın bozulmasını önlemek için, yalnızca blok zincirleri merkezden uzaklaştırılmakla kalmaz, aynı zamanda kriptoloji kullanılır. Blockchain ağları, ağın bütünlüğünü korumak için bir teşvik olarak kripto para birimleri üretir. Birçok kripto para birimi, hisse senedi gibi borsalarda işlem görür. Hatta bir borsadan paranın çıkarılması değerinde ciddi düşüşler yapabilmektedir. İşin teknolojisinden bağımsız olarak borsalar da fiyatlar üzerindeki gizli bir otorite olarak görülebilir.

Peki neden bu kadar çok kripto para var? Çünkü kripto para birimleri, farklı çalışma formülleri geliştirmiştir. Kitapta Algorand'ın formülüne detaylı bakacağız.

Temel olarak yazılım, donanımın çalışması için ödeme yapar. Yazılım, blockchain protokolüdür. Bilinen blok zinciri protokolleri arasında Bitcoin, Ethereum, Algorand vs. bulunur.

Blockchain Uygulamaları

Blockchain uygulamaları, ağın hakem, kriptolojinin hakemin düdüğü olduğu fikri etrafında inşa edilmiştir.

Bu sistemlerde bilgisayar kodu kanun haline gelir ve kurallar ağ tarafından yazıldığı ve yorumlandığı gibi yürütülür. Bilgisayarlar, insanlarla aynı sosyal önyargılara ve davranışlara sahip değildir. İstisnai bir durum olarak belirtmek gerekir ki akıllı kontrat önyargılı bir şekilde yazılmış olabilir. Fakat kodlar herkes tarafından görülebildiği için bu sorun önemli ölçüde çözülmüştür.

TÜBİTAK'ın yayınladığı listeye göre blokzincirin genel olarak uygulama alanları şunlardır:

- Bankacılık
- FinTech
- Para Transferleri
- Değerli Belgelerin Yaratılması ve Saklanması



- E-Ticaret ve Ödemeler
- Hisse Senetleri ve Borsalar
- E-Noter
- Kişiden Kişiye Borçlanma ve Dağıtık Yapılı Kredi Sistemleri
- Bağış Sistemleri ve Mikro Ödemeler
- Bulut Bilişim ve Güvenli Bulut Depolama

Ama bunlar inanın çok küçük bir kısmı. Ben de uygulama alanı olmayan bir şeyin olamayacağını düşünenlerdenim. Çünkü içinde güven olmayan hiçbir şey yok. Evden çıktığınızda havayı koklarken, evde su içerken bile gizli bir akıllı sözleşme zihninizde var oluyor.

Şimdi gelelim Algorand'a.

İKİNCİ BÖLÜM

MERHABA ALGORAND



İlk bölümde bahsettiğimiz gibi Dağıtılmış Defter Teknolojileri merkezi bir nokta tarafından korunmadan herkese açık olarak erişilebilen ve değiştirilebilen, dayanıklı bir veri dizisi olarak düşünülebilir. Dağıtılmış defterler, modern bir toplumun işleyiş biçiminde devrim yaratacak niteliktedir. Değer aktarımına farklı bir bakış açısı getiren bu defterler ödemelerden varlık transferlerine kadar birçok konudaki geleneksel işlemi güvence altına alabilirler. Aracıları kaldırabilir ve güven içinde yeni bir fikir birliği ortaya çıkarabilirler. Bitcoin gibi kriptopara projeleri de bu amaçla yola çıkmıştır. Fakat geçen zaman içerisinde mevcut dağıtılmış defter projelerinin zayıf bir şekilde ölçeklendiği ve sahip oldukları potansiyellerini gerçekleştirmedikleri görülmüştür. Bu çalışmada Algorand ismi verilen teknoloji ele alınmıştır. Algorand güvenli, merkezsiz ve ölçeklenebilir bir blokzincir projesi olarak ön plana çıkmaktadır.

Algorand

Algorand, merkezi olmayan projeler ve dağıtık uygulamalar yoluyla, şeffaf bir sistem oluşturmayı amaçlayan blockchain projesidir. Sistemde iş ispatı olarak Proof Of Stake'in gelişmiş bir versiyonu olan Pure Proof of Stake kullanılmaktadır. Algorand projesinin ana geliştiricisi Silvio Micali'di. Şifrebilim ve bilgi güvenliği konulu çalışmalarıyla tanınan Micali, Massachusetts Teknoloji Enstitüsü'nde profesör olarak çalışmaktadır. 1993'te verilmeye başlayan Gödel Ödülü'nü kazanan ilk bilim insanlarından biri olan Micali 2012 yılında Turing Ödülü'ne değer görülmüştür. Projenin arkasındaki ismin kriptografi alanındaki bilinirliği Algorand'a olan ilgiyi artırmaktadır.



Algorand ile ilgili ilk araştırma makalesi “Algorand: Scaling Byzantine Agreements for Cryptocurrencies” ismiyle Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos ve Nickolai Zeldovich tarafından 2017 yılında bir konferansta yayınlanmıştır.

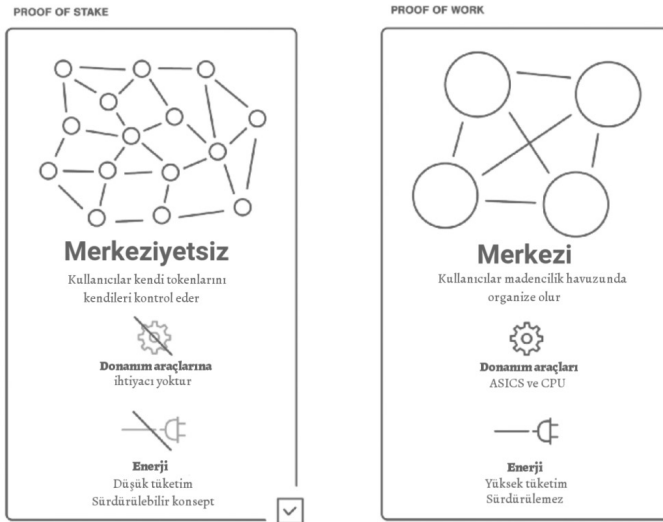
Bu makalede Algorand’ın Bizans hata toleransı veya Bizanslı Generaller Problemi olan konuya yeni bir yaklaşım yaptığı, bu sayede ölçeklenebilen ve güvenli bir sistem inşa edilebileceği vurgulanmaktadır. Bu yaklaşımla çok fazla kullanıcıda dahi sistem bir dakikadan daha az bir sürede değer transferlerini gerçekleştirebileceği belirtilmektedir.

Pure Proof-of-Stake

Algorand’ın kullandığı protokol olan Pure Proof-of-Stake(PPoS) fikir birliği mekanizması veya algoritma olarak da isimlendirilir. Bu protokoller sayesinde blokzincirdeki işlemlerin üçüncü taraf gerekmeden hızlıca onaylanması sağlanır. PPoS’u daha iyi anlayabilmek için PoW ve PoS’a yakından bakılabilir.

Kriptoparaların atası kabul edilen Bitcoin’de değer aktarımlarının, işlemlerin ispatlanması için Proof-of-Work ismi verilen bir sistem kullanılmaktadır. Bu sistemde katılımcılarının ağı dâhil olabilmesi için emek harcamalarını yani iş yapmalarını gerektirir. Bu iş tam olarakta madenciliktir. Katılımcılar ileri düzey cryptographic (kriptografik) şifreleri işlemciler ile çözerek yapılan transferleri onaylar ve kaydeder. Kaydedilen bu veriler halka açık olarak kaydedilir ve herkesin görebileceği işlemlerdir. Verilen bu emek neticesinde de kriptografik yapbozu çözen madenci ödül olarak belirlenen miktarda kriptopara kazanır. Fakat bu sistemde işlem hızlarının düşük olması, yoğunluk yaşandığında artan komisyon ücretleri ve madencilikte gerçekleşen elektrik tüketiminin devasa boyutlara ulaşması nedeniyle alternatif fikir birliği mekanizmaları düşünülmüştür. İlk olarak

2011 yılında BitcoinForum’unda bir kullanıcı Proof-Of-Stake ismi verdiği bir model önermiştir. Önerilen PoS da diğer mutabakat algoritmaları ile aynı amacı paylaşmaktadır. Amaç dağıtık mutabakat sağlamak ve bunu yaparken kullanıcıları başkalarının işlemlerini doğrulamak için teşvik etmek. Proof of Stake algoritması sözde rastgele seçim yöntemiyle düğümlerden birisini sonraki bloğu doğrulamak için görevlendirir. Seçim zamanı bahis yaşı, rastgele olması ve düğümün toplam varlığı gibi faktörlerin toplamı dikkate alınır. Adım adım düşünülürse, ilk olarak Algoritma bloke hesabında kripto parası çok olanı seçer. İşlemi ispatlama işini bu hesap yapar. Ardından madenciye karşılık gelen bu doğrulayıcı için bir coin oluşturulmaz, fakat ödül olarak işlem ücretini alır. Ardından coin yaşı sıfırlanır. Fakat bu sistemde çok parası olanın daha şanslı olması iki ana sorun doğurmaktadır. Bunlardan ilki demokratikleşmeyi ortadan kaldırmakta ve gitgide merkezi bir yapı haline getirmektedir. İkincisi ise siber saldırılar için çok parası olan cüzdanları açık hedef haline getirmekte ve bir güvenlik zafiyeti oluşturmaktadır.



Hesapların kime ait olduğu normalde gizli olsa da, tozlama saldırısı gibi sosyal mühendislik yöntemleriyle keşfedilebiliyor. Bu güvenlik zafiyetini ortadan kaldırmak ve demokratikleşmeyi sağlamak için Algorand sistemi PPOs ismi verilen sistemi geliştirmiştir.

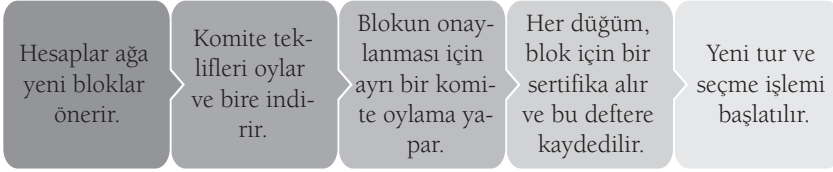
PPOs protokolünde her kullanıcı yeni blok seçiminde etkili olabiliyor. Bu yolla hem demokratikleşme hem de güvenliğin sağlanması amaçlanıyor. Kullanıcılar blok keşiflerinde ve tekliflerinde rastgele ve gizlice seçiyor. Algorand’ın PPOs algoritması, tüm ekonominin güvenliğini küçük bir alt kümenin doğruluğuna değil ekonominin çoğunluğunun doğrulu-



ğuna bağlamayı amaçlıyor. Paranın çoğunluğu doğru ellerde olduğunda sistem güvenli olarak tanımlanabiliyor. Diğer protokollerde ekonominin küçük bir grup tüm ekonominin güvenliğini belirleyebiliyor ve bu sebeple birkaç kullanıcıdan oluşan bu grup bütün kullanıcıların işlem yapmasına mani olabiliyor. Algorand'da, paranın küçük bir kısmına sahip olan bir grubun tüm sisteme zarar vermesi imkânsız denilebilir. Eğer bir grup paranın çoğunluğuna sahip olursa paranın alım gücü düşer ve bu da doğal olarak paranın değerinin düşmesine sebep olur. Bu nedenle çoğunluğa sahip oldukları için zarar ederler ve bu hareket de pek mantıklı olmaz.

Algorand üzerinde blok önerme işlemlerinde şu işlem sırası izleniyor.

- Blok Teklifi: Hesaplar ağa yeni bloklar önerir
- Soft (Yumuşak) Oy: Komite teklifleri oylar ve bire indirir
- Oyu Onayla: Bloğu onayı için ayrı bir komite oylama yapar
- Her düğüm, blok için bir sertifika alır ve bu deftere kaydedilir
- Yeni tur başlatılır ve blok önerenler ve seçmenlerle süreç yeniden başlar



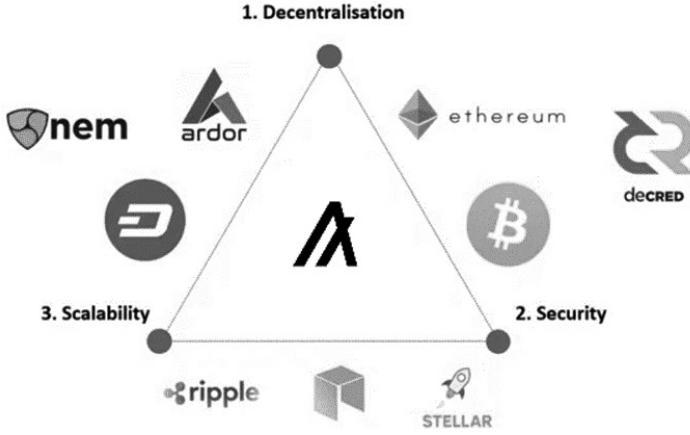
PPoS üzerinde ödül sistemi de PoS'a göre değiştirilmiş durumda. PoS sistemlerindeki gibi stake yaparak varlıkları belirli süreler boyunca kilitlemeye gerek olmuyor.

PPoS, Algorand'ın en önemsedığı noktası diyebiliriz. Bu algoritma ile Blockchain Üçlemesi diyebileceğimiz "Blockchain Trilemma" için de problemi çözdüğünü iddia ediyor.

Blokzincir'in Üçlemesi: Güvenlik, Merkezsizlik, Ölçeklenebilirlik

Ethereum kurucularından Vitalik Butelin bir blokzincirin aynı anda hem güvenli hem merkezsiz hem de ölçeklenebilir olamayacağını, bu üç durumdan en fazla ikisini seçmesi gerektiğini dile getirmiştir. Fakat Algorand Butelin'in bu düşüncesini çürüttüğünü iddia etmektedir.

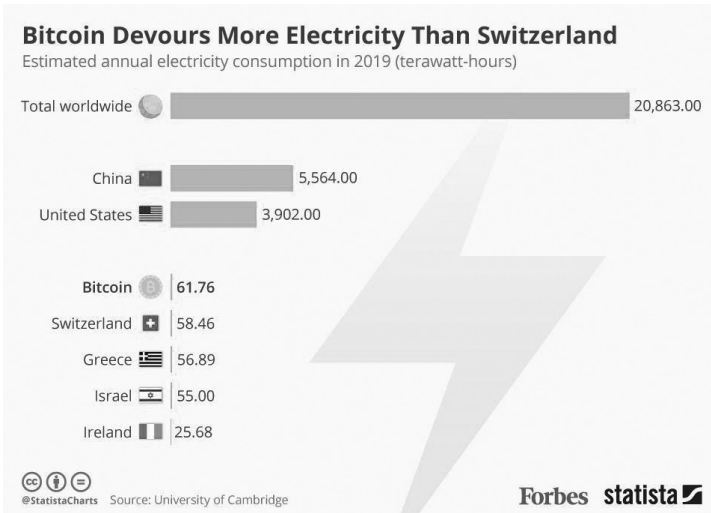
Görselde bu konuda güvenlik, merkezsizlik ve ölçeklenebilirlik noktalarına göre kriptoparaların dağılımına yer verilmiştir.



Bu üç kavram şu şekilde tanımlanabilir.

Merkezi olmayan: Merkezi bir kontrol noktasına dayanmayan bir blok zinciri sistemi oluşturmaktır. Bu zaten Blokzincir'in çıkış misyonunda vardır. Geleneksel finasta işler tamamen merkezi giderken Blokzincir teknolojileri bunu bir anlamda da demokratikleşme yoluna götürerek ademi merkeziyet mottosunu benimsemektedirler.

Ölçeklenebilir: Merkezi olmama durumun bir süre sonra ölçeklenebilirlik problemini de yanında getirebilmektedir. Ölçeklenebilirlik bir blokzincir sisteminin giderek artan miktarda işlemi idare etme yeteneği olarak tanımlanmaktadır. Bitcoin'in işlem (transaction) yeteneğinin Visa ile karşılaştığında kapasitesinin düşük olması veya İzlanda gibi bir ülke kadar elektrik sarfiyatına yol açması da yeterince ölçeklenebilir olmamasının getirdiği sorunlardandır.





Güvenli: Blockchain sisteminin beklendiği gibi çalışabilme, kendini saldırılara, hatalara ve diğer öngörülemeyen sorunlara karşı savunma yeteneği de güvenlik olarak tanımlanıyor. Merkezi bir yapı kurulduğunda ölçeklendirme problemini aşarken güvenlik ihmal edilirse “Blockchain Trilemma” çıkmazına tekrar düşülmesine sebep olmaktadır. Güvenlik noktasında Algorand “mnemonic” ismi verilen 25 kelimeden oluşan şifreler kullanmaktadır. Ayrıca çoklu imza hesapları ile de bir güvenlik adımı atmaktadır.

Algorand Üzerinde Tekli ve Çoklu İmzalı StandAlone Hesap Oluşturma

Algorand’ın sisteminin işleyişi ve alternatif blokzincirlere göre öne çıktığı noktalar önceki bölümlerde anlatılmıştır. Bu bölümde ise daha teknik bir çalışma yapılarak blokzincir üzerinde hesaplar oluşturulacaktır.

Algorand StandAlone ismini verdiği bir hesap türü oluşturmuştur. Bu hesabın bir tane public adresi bir de private keyi bulunmaktadır.

Private key kısmını 25 kelimelik anımsatıcıdan oluşturmuştur. 25 rastgele kelime kullanımı bir tür güvenlik önlemidir. Bu kelimeler, sırasıyla, mutlaka not alınmalıdır. Benzer bir sistem ethereum için Metamask eklentisinde de bulunmaktadır.

StandAlone hesaplar “bağımsız hesap” olarak tanımlanabilir. Bir diske bağılılığı yoktur dolayısıyla farklı donanımlardan kullanılabilir, ayrıca hızlıca oluşturulabildiği için geliştiriciler için de büyük bir kolaylık sağlamaktadır.

Algorand’ın mobil cüzdanı (Android, iOS) da StandAlone hesap kullanmaktadır. Burada da hesapları mobil cüzdana aktarmak için 25 kelimelik anımsatıcıdan faydalanılmaktadır.

StandAlone hesabın dezavantajı ise nispeten güvenlidir. Eğer 25 kelimelik anımsatıcı güvenliğini zayıf buluyor ve şifrelenmiş bir disk içinde hesabınızı tutmak istiyorsanız kmd denilen ve donanımla beraber çalışan farklı bir hesap türü kullanılabilir. Bu hesapları Python programlama dili kullanarak oluşturmak için ilk olarak;

pip install py-algorand-sdk

diyerek kütüphane yüklenir.

Ardından proje için bu kütüphaneden account ve mnemonic kısımları; `from algosdk import account, mnemonic` koduyla alınır. Sonrasında `algorand_anahtarlarimi_olustur()` ismini verdiğimiz fonksiyonda `account.generate_account()` komutuyla private key ve adres oluşturulur.



```
def algorand_anahtarlarimi_olustur():
    private_key, address = account.generate_account()
    print("Adresim: {}".format(address))
    print("\n")

    print("Hatırlatıcı Kelimelerim:{}".format(mnemonic.from_private_
    key(private_key)))

algorand_anahtarlarimi_olustur()
```

Bu şekilde fonksiyonu oluşturduk ve çalıştırdık. Çıktımız aşağıdaki şekilde oluyor.

Adresim: 6KCPU5QW5PG3B3MOFF5ZHUFHIXOJGT5BVZM5RKS7JH5VEJGH2ITLFYXSE

Hatırlatıcı Kelimelerim: need crisp require bachelor tourist reject side leaf retreat calm forum rack chef saddle purchase exit antique almost off analyst speed noodle become above oxygen

Oluşturulan hesabın doğruluğu Algo Explorer sitesinden kontrol edilebilir.

Görselde görüleceği gibi hesap aktif şekilde durmaktadır. Henüz hiç transfer olmamıştır ve içinde para bulunmamaktadır. Tekli hesabın dışında çoklu imzaya sahip hesaplar da oluşturulabilmektedir. Gerek Bitcoin’de, gerek Ethereum’da aslında çoklu imzalı cüzdan teknolojisi kullanılmaktadır. Çoklu imza kullanımı anlamak için bir kasa ve anahtar örneği verilebilir.

Bir kasa olduğu ve aynı anda kullanılması gereken 3 farklı anahtar olduğunu düşünelim. Her birimizde de bir anahtar var. Eğer o kilidi açmak istiyorsak üçümüzün bir araya gelerek anahtarlarımızı kullanmamız lazım, aksi takdirde kasa açılmayacak. Bunu kripto para teknolojilerinde ortak yönetilen bir fon için düşünülebilir. Tabi sadece tüm anahtar sahiplerinin dâhil olduğu senaryolar yoktur. Şöyle bir durum da olabilir. 9 kişide anahtar var ve 5 kişi, yani çoğunluk transferi kendi anahtarıyla onaylarsa olur.



Borsalarda ve çok büyük bütçeli projelerde bu anahtarlar popülerdir. Algorand teknolojisinde Bitcoin'den farklı olarak adresin çoklu imza hesabı olup olmadığını dışardan biri bilemiyor. Bir kat daha güvenlik artırılmış gibi düşünebilirsiniz.

Avantajı dediğimiz gibi güvenlik. Dezavantajı ise kolaylıktan taviz vermektir. Tekli imza hesaplarının kullanımı daha pratiktir. Python ile çoklu imza oluştururken ilk olarak `algosdk` kütüphanesinden `account` ve bu sefer farklı olarak `transaction` kısımlarını içe aktarılmaktadır.

İlk olarak 3 farklı tekli hesap oluşturulabilir. Bu sayı daha farklı da olabilir.

```
from algosdk import account, transaction
```

```
# üç hesap oluşturalım
```

```
anahtar_1, hesap_1 = account.generate_account()
anahtar_2, hesap_2 = account.generate_account()
anahtar_3, hesap_3 = account.generate_account()
print("Hesap 1:", hesap_1)
print("\n")
print("Hesap 2", hesap_2)
print("\n")
print("Hesap 3:", hesap_3)
print("Anahtar 1:", anahtar_1)
print("\n")
print("Anahtar 2:", anahtar_2)
print("\n")
print("Anahtar 3:", anahtar_3)
print("\n")
```

Çıktı görseldeki şekilde olmaktadır.

```
Hesap 1: J6WIG5AHUXW2WI3KEPUJJTZSGQIPI6XYTJKYPZRKZI3ZWIDR56OZVANINY
```

```
Hesap 2 APM2E4FOM3I6MOXVZPSGMGWCV6HJ3YGSTFUZFRIB77I7QRYGDBD2CM6Z2A
```

```
Hesap 3: PNOQAZ66IVUCTJ6HV76VHX7RSLZDNWZ6CVZK77G4Z5OYYNJOVROHP4E64
Anahtar 1: Yk6CajbC5MpLtVUjAYRLYmQ8VHxQl6fnbo00C+8lxlPrIN0B6XtqyNqI
+iUzzI0EPR6+JpVh+YqyjebIHHvnQ==
```

```
Anahtar 2: j0anJXtdvCXAfh5hXkqmkpKFvM1fGkqV7RhsAF1rpF0D2aJwrmbR5jr1y
+RmGsKvp3g0plpksUB/9H4RwYYRw==
```

```
Anahtar 3: TMyxdIBbTYZ8JL0DtTCODI68OZJAvLp14WGbPUDZYyV7XQ8n3kVoKafHr/1qnV
+M15G22fCr1X/m5nrsYal1Yg==
```

Hesapları ve anahtarlar oluşmuştur. İkinci adımda teklif hesaplar çoklu imzalı bir hesap haline getirilecektir.

Bu adımda çoklu imza hesap versiyonu, kaç kişinin transfere onay vereceği ve bunların hangi hesaplar olduğu bilgisini girilmektedir.

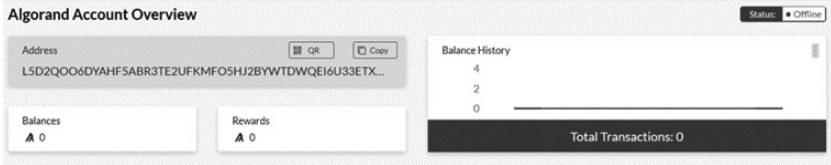
```
#çoklu imza hesabı yapma
version = 1 #çoklu imza versiyonu
onay_veren_hesap=3 #onay için kaç imza gerekli
msig = transaction.Multisig(version,onay_veren_hesap,[hesap_1,hesap_2,hesap_3])
print("Çoklu İmza Hesap Adresi:", msig.address())
```

Görselde çıktı görülmektedir.

Çokluimza Hesap Adresi:

L5D2Q006DYAHF5ABR3TE2UFKMF05HJ2BYWTDWQE16U33ETXKQYBYPXQGI

Hesap oluşmuştur. Hesabın kontrolü Algo Explorer üzerinden sağlandığında bu hesabın tekli mi yoksa çoklu bir hesap mı olduğu belli değildir. Burada gizlilik uygulanmaktadır.



Bu örnek uygulamada onay için 3 değil 2 hesap da denilebilir. 2/3 oranı sağlandığı için işlemi gerçekleştir anlamında bu tarz seçimler de yapılabilir.

Algorand Cüzdanı

Algorand blokzincir teknolojilerinde önemsenen bir nokta da Algorand Cüzdanı'dır. Türkçe dil desteği de olan bu cüzdan İos ve Android platformlardan indirilebilmektedir. Cüzdanda varlıklar görüntülenebilmekte, PPOS üzerinden alınan ödüller görülebilmekte, MainNet'e ek olarak geliştiriciler için olan TestNet'e geçişler yapılabilmektedir. Bluetooth entegrasyonu ile fiziksel cüzdanlar ile entegrasyon sağlanarak güvenlik sağlanabilmektedir. Standalone hesap oluşturma bölümünde ele alınan 25 kelimelik özel anahtar da cüzdan üzerinden değiştirilebilmektedir. Bu değişim işlemi esnasında cüzdan adresinin değişmemesi kullanıcılara esneklik sağlamaktadır.





Algorand Ortamı Kurulumu

Bu yazıda Algorand üzerinde bir uygulama oluşturmanın ne anlama geldiğine ve nasıl hızlı bir şekilde bu geliştirme ortamına giriş yapabileceğimizi ele alacağız.

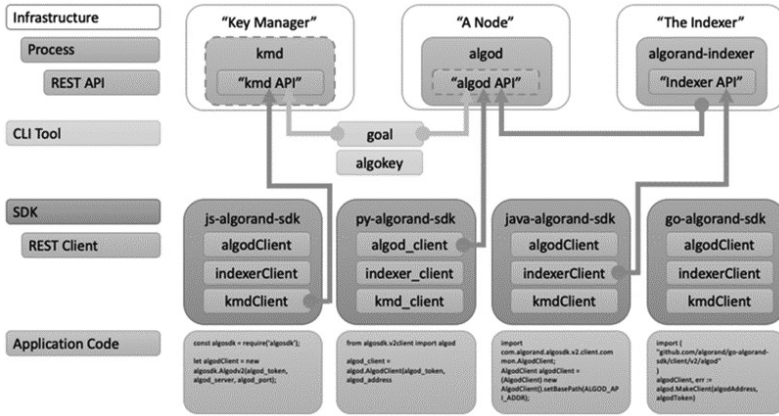
Algorand üzerinde bir uygulama oluşturmak, uygulamanızın Algorand blokzincirinden doğrudan veya dolaylı olarak okuduğu veya zincire yazdığı anlamına gelmektedir.

Burada “yazmak” ile kastedilen şey daha sonrasında bir blok içinde onaylanacak bir işlem yapmakla eş anlamlıdır.

Blok zincirinden okumak ise önceki bloklarda onaylanan işlemleri geri okumak anlamına gelir.

Algorand Mimarisi

Bir uygulama inşaa ederken nasıl bir mimari kullanacağımız, nasıl bir yol izleyeceğimiz son derece önemlidir. Algorand bu noktada “esnek mimari” avantajıyla ön plana çıkmaktadır. Esnek mimariden kasıt ilerleyen kısımlarda detaylarını göreceğiniz şekilde geliştiriciye farklı yol haritaları sunmasıdır.



Yukarıdaki görselde mimari detaylı olarak görülmektedir. Infrastructure olarak ifade edilen alt yapılar üçe ayrılmaktadır. Bunlar Key Manager, A Node ve The Indexer şeklindedir. Bunları yakından inceleyelim.

Key manager: Güvenlik olarak kullanılan anahtarların yönetildiği kısım.

A node: Key manager ve “a node” beraber kullanılarak goal ve algokey geliştirici araçları işbirliği yaparken, A node ve The Indexer da beraber kullanılabilir.

CLI Tool: Komut sistemiyle işlemler yapmak. Blokzincirde yazmak veya blokzincirden okumak.



SDK: Javascript, Python, Java veya Go programlama dillerinden seçilen birinin geliştirme kütüphanesini kullanarak uygulamayı geliştirmek. Biz çalışmamızda Python SDK'sı ile geliştirme yapacağız.

Bu bilgiler ışığında Algorand blok zincirini tekrar tanımlayalım. Algorand Blok Zinciri blokların geçmişini ve içindeki işlemleri doğrulamaya dayalı olarak her biri yerel durumlarını koruyan dağıtık bir düğüm sistemidir.

Veriler, arka plan programı içinde uygulanan ve genellikle düğüm yazılımı olarak adlandırılan mutabakat protokolü tarafından tutulur. Bu yolda ilk adım algod istemcisidir. **Algod istemcisi**, geliştiriciler için uygulamaların temel katmanıdır.

Bir uygulama, Algorand blok zincirine bir **algod** istemcisi aracılığıyla bağlanır. Algod istemcisi, etkileşim kurmayı planladığınız ağa bağlı bir Algorand düğümünden geçerli bir **algod REST uç nokta IP adresi** ve **algod belirteci** gerektirir. Algod ile döndürebileceğiniz bilgilere detaylı olarak <https://developer.algorand.org/docs/reference/rest-apis/algod/> adresi üzerinden ulaşabilirsiniz.

Araç Seçimi

Algorand üzerinde uygulama geliştirmeye başlamadan önce bir araç seçimi yapmanız gerekmektedir. Hali hazırda 3 farklı araç vardır. Bunlar SDK, CLI ve Dizin Oluşturucudur. Bu araçları tanıyalım.

Yazılım Geliştirme Kitleri (SDK)

Algorand, uygulama geliştirmek için resmi olarak dört SDK'yı destekler: Javascript, Java, Python ve Go. Ek olarak, Topluluğun Sağladığı SDK'lar geliştirme erişimini genişletir. Topluluk aktif olarak farklı programlama dillerine genişlemenin nasıl olabileceğini tartışmaktadır.

Komut Satırı Arayüzü (CLI) Araçları

Algorand Algorand düğüm yazılımı ile paketlenmiş üç komut satırı yarar sağlar: goal, kmd ve algokey. goal bir düğümü çalıştırmak için birincil araçtır ve ayrıca anahtarları yönetmek, işlemleri imzalamak ve göndermek, varlıklar oluşturmak ve SDK'larda bulunan birçok aynı veya benzer işlevi gerçekleştirmek için işlevler içerir. Bir uygulama oluşturmaları gerekmede de, düğümleri çalıştıran geliştiriciler, goal test ve doğrulama sırasında tamamlayıcı rol üstlenir. Özel ağları kullanarak daha gelişmiş test ortamları kurmak goal için gereklidir.

kmd Algorand Key Management arka plan algokey programı için CLI'dir ve Algorand hesapları oluşturmak ve işlemleri imzalamak için bağımsız bir yardımcı programdır. Genellikle güvenli anahtar imzalama için



bir çevrimdışı istemci olarak kullanılır. Bu iki araç başlamak için zorunlu değildir, kullanımlarının ayrıntıları ileride açıklanacaktır.

algod ve kmd her ikisi için de kullanılabilir REST API'leri de vardır.

Dizin Oluşturucu

Algorand, Algorand blok zincirinden işlenmiş blokları okuyan ve aranabilir ve dizine eklenmiş işlemlerin ve hesapların yerel bir veritabanını tutan bağımsız bir daemon algoritması ve indeksleyici sağlar. Uygulama geliştiricilerin hesaplar, işlemler, varlıklar vb. Üzerinde zengin ve verimli sorgular gerçekleştirmesini sağlayan bir REST API mevcuttur.

Ağ Seçimi

Farklı blokzincir teknolojilerinde olduğu gibi Algorand üzerinde de uygulamalarınızı test edebileceğiniz bir kaç farklı ağ vardır. Herhangi bir protokol sürümünü kullanarak kendi özel ağlarınızı oluşturmak için işlevsellikle eşleştirilmiş üç genel Algorand Ağı bulunur.

Bunlardan ilki olan MainNet, Algorand'ın yerel para birimi olan Algo da dahil olmak üzere gerçek değerli varlıklara sahip birincil Algorand Ağıdır. Ana ağ diyebiliriz.

TestNet, MainNet'in protokol (yani yazılım) sürümü açısından kopyası gibidir ancak bir (faucet) musluk aracılığıyla erişilebilen test Algos'a ve farklı bir oluşum bloğuna sahiptir, bu da hesapların durumunun ve fonların dağıtımının farklı olduğu anlamına gelir. Geliştiriciler için bir tür test noktası işlevi görür.

BetaNet, ilk test için yeni protokol düzeyindeki özelliklerin yayınlanacağı yerdir. Bu nedenle, kalite ve özellikler nihai olmayabilir ve protokol yükseltmeleri ve ağ yeniden başlatma olayı yaygındır.

Bu noktada , yani ağ seçme sürecinde, Algorand'ın tavsiye ettiği bir kullanım usulü bulunmaktadır.

Daha detaylı olarak tablodan bakalım. Buradaki protokol sürümünde BetaNet'in Gelecekte devreye alınacak olması önemlidir. Eğer uygulamanızda sadece BetaNet üzerinde mevcut özellikleri devreye alırsanız, MainNet üzerinde çalışırken sorun yaşayabilirsiniz. Genesis Dağıtım denilen kısım ise yapılan işlemlerin ağda benzersiz olduğu detayıdır. Yapılan işlemler her 3 zincirde de kalıcı olarak benzersiz olmaktadır. Algo Erişilebilirliği, MainNet üzerinde satılıktır. Yani burada yaptığınız hizmeti satabilirsiniz. Fakat TestNet veya BetaNet üzerinde satım yapamazsınız. Ağ güvenilirliği olarak MainNet ana ağ olduğu için en kararlı ağdır. Diğer ağlarda yeniden başlatmalar olduğu için daha kararsız bir hal olabilmektedir. Özellikle BetaNet ağı sürekli gelişim içinde olduğundan, bir laboratuvar gibi deneysel çalışmaların alanıdır. Bu da en kararsız ağ olmasına yol açmaktadır.



	MainNet	TestNet	BetaNet
Protokol Sürümü	Güncel	Güncel	Gelecek
Genesis Dağıtım	Benzersiz	Benzersiz	Benzersiz
Algo Erişilebilirliği	Satılık	Mustuksuz	Mustuksuz
Ağ Güvenilirliği	En Kararlı	Çok Kararlı, ancak yeniden başlatmalar mümkündür	Deneysel; sık yeniden başlatma

Algod Adresi ve Tokeni Elde Etme

Araç seçimini ve ağ seçimini yaptıktan sonra bir algod adresi ve tokeni elde etme kısmına geçelim. Algorand'ın tanıdığı esneklik burada da devam etmektedir. Her birinin artısı ve eksisi olan üç farklı yoldan birini seçerek algod REST uç nokta IP adresi ve erişim belirteci alabilirsiniz. Burada hedefinizin ne olduğu önemlidir. İster üçüncü parti bir hizmet kullanabilir, ister Docker Sandbox kullanabilir veya kendi düğümünüzü çalıştırma yoluna gidebilirsiniz.

Bu seçimlere yakından bakalım ve ardından karşılaştırmalı bir tablo yapalım.

Üçüncü Parti Hizmet Kullanımı: Bu yöntem, yalnızca SDK'ları veya algod RESTful arabirimini kullanmayı planlıyorsanız ve olabildiğince hızlı bağlanmak istiyorsanız önerilir. Üçüncü taraf bir hizmet, bir düğümü çalıştırır ve bu düğüme kendi API anahtarları aracılığıyla erişim sağlar. Kayıt sırasında hizmet size bir algod adresi ve algod jetonunuzun yerini alacak bir API anahtarı sağlar. Aşağıdaki görselde kullanabileceğiniz 2 platform gösterilmiştir. PureStake ve AlgoExplorer API hizmeti sunmaktadır. Elbette bu sayı ilerleyen zamanlarda artabilir, internet siteleri değişebilir. Fakat genel olarak mekanizmayı bilmek faydalı olacaktır.

İsim	Açıklama
 PureStake API Hizmeti	PureStake'in API hizmeti, Algorand ağına hızla çalışmaya başlamayı kolaylaştırır. Hizmet, geliştiricilere yerel Algorand REST API'lerine kullanımı kolay erişim sağlamak için PureStake'in mevcut altyapı platformunu temel alır.
 AlgoExplorer Rand Labs API Hizmeti	Rand Labs'in API Hizmetleri. Algod işlemleri için API Uç Noktası ve AlgoExplorer API

Docker Sandbox Kullanımı: Docker, uygulamalarınızı hızla derlemenize, test etmenize ve dağıtmanıza imkân tanıyan bir yazılım platformudur. Sandbox ise bu platform içerisinde izole edilmiş bir ortam gibi düşünülebilir. Docker Sandbox kullanımı üçüncü parti uygulama kullanımına göre biraz daha yavaştır. İşlem saniyeler değil dakikalar içerisinde olur. Fakat üçüncü parti uygulamaya göre önemli bir avantaja sahiptir. Bu platformda goal, kmd, algokey geliştirici araçlarının tamamına erişim sağlayabilirsiniz.



Kendi Düğümünüzü Çalıştırma: Kendi düğümünüzü çalıştırmak yöntemler içerisinde kurulumu en uzun süreni fakat en güçlü altyapıya sahip olduğunuz yöntemdir. Tüm goal, kmd, algokey geliştirici araçlarına erişim sağlanmasının yanında kurulum için üretime hazır bir ortam sağlanır. Bu yöntem düğümünüzün ve yapılandırmanın tam kontrolünü size verir. Kurulumdan sonra REST uç noktanızın IP Adresi ve token bilgisini aşağıdaki komutlarla öğrenebilirsiniz.

IP adresi:

```
$ cat $ALGORAND_DATA/algod.net
```

Algod token bilgisi:

```
$ cat $ALGORAND_DATA/algod.token
```

Şimdi 3 yöntemi tablo üzerinden karşılaştıralım.

Yan Yana Karşılaştırma

	Üçüncü taraf bir hizmet kullanın	Docker Sandbox'ı kullanın	Kendi düğümünüzü çalıştırın
Zaman	Saniye - Sadece kaydolun	Dakika - Yakalama olmadan bir düğümü çalıştırmayla aynı	Günler - düğümün yakalanması için beklemeniz gerekir
Güven	1 parti	1 parti	Kendin
Maliyet	Genellikle geliştirme için ücretsizdir; üretimdeki oran limitlerine göre ödeme yapın	Değişken (ücretsiz seçenekle) - düğüm türlerine bakın	Değişken (ücretsiz seçenekle) - düğüm türlerine bakın
Özel Ağlar	✗	☑	☑
goal , algokey , k md	✗	☑	☑
Platform	Değişir	Mac os işletim sistemi; Linux	Mac os işletim sistemi; Linux
Üretime Hazır	☑	✗	☑

Ortam ile ilgili tüm seçimler bu yol haritasıyla belirlendikten sonra kodlama kısmına geçilebilir.

ÜÇÜNCÜ BÖLÜM

PYTEAL İLE AKILLI KONTRATI OLUŞTURMA

Bu bölümde Python üzerinde Pyteal kurulum işlemlerinin nasıl yapılacağına bakacağız. İlk olarak pip isimli paket yönetim sistemidir kurmamız gerekiyor. Python dağıtımlarının çoğu önceden yüklenmiş pip ile gelmektedir.

Bundan dolayı Python kurulumu yaptıktan sonra pip kurulumu için ekstra bir şey yapmıyoruz. Pyteal kurulumu için ilk olarak;

```
pip3 install py-algorand-sdk
```

diyerek Algorand tarafından Python programlama dili için hazırlanan SDK'in kurulumunu yapıyoruz.

Ardından da;

```
pip install pyteal
```

diyerek pyteal kütüphanesini kuruyoruz.

Sistemin sorunsuz kurulduğunu Başlat>Cmd>Python>from pyteal import * diyerek kontrol edebiliriz.

```
>>> from pyteal import *  
>>>
```

Görülebileceği gibi kodumuz sorunsuz çalıştı.

Şimdi birazcık teorik bilgiyle devam edelim. Akıllı sözleşmeler veya daha bilinen ismiyle akıllı kontratlar şu işe yarıyor. Aradaki aracıyı ortadan kaldırarak her iki tarafından kriptoloji ile düzenlenmiş bir dijital anlaşmaya güvenmesini sağlıyor. Biz işte burada bu dijital anlaşmayı yazıyoruz.

Şöyle bir düşünelim. Aracıların olmaması hangi alanlarda neleri değiştirir. Elbette ilk akla gelen tapu, nüfus gibi kayıtlar olacaktır. Fakat bunların ötesinde bugün itibarıyla dahi güvenilir olmadığı için yeterince çalışmayan, insanların ilgisini çekmeyen sistemler var. Örneğin kitlesel bağış mekanizmaları. Halen daha toplumun yüzde kaç bu sistemleri kullanıyor? Bunların ötesinde bir de henüz elektronik işlemleri tamamen merkezi ola-



rak düşündüğümüz, aracısız bir konsept hayal edemediğimiz için ortaya çıkmamış bir potansiyel de var. İşte bu nedenlerle akıllı kontratlar son derece önemlidir.

Algorand'ta akıllı kontratlar, Algorand Smart Contract (ASC) olarak geçer.

ASC'ler, İşlem Yürütme Onay Dili (TEAL) adı verilen bayt kodu tabanlı bir dilde yazılır. TEAL kısaltması, Transaction Execution Approval Language kelimelerinin baş harflerinden gelmektedir ve Algorand projesi kapsamında yazılmış bir dildir. Bu dilde genel mantık blokzincir ağına yazılan bir işlemin sistemin kendi mantığına göre analiz edilerek onaylanıp onaylanmadığını ve sırasıyla doğru ya da yanlış onaylı ya da onaylanmamış olarak döndürülmesi esasına dayanmaktadır.

TEAL, işlemleri analiz edebilir ve onaylayabilir ancak işlemleri oluşturamaz veya değiştiremez ve sonuç olarak yalnızca doğru veya yanlış döndürür.

Şimdi ilk projemize geçiyoruz. İyi anlaşılması için kodumuzu satır satır inceleyeceğiz.

from pyteal import * diyerek Pyteal içerisindeki herşeyi buraya getir dedik.

Şimdi sadece para çekmeye izin veren bir kontrat yapalım. İlk olarak;

```
def bank_for_account(receiver):
```

diyerek bir fonksiyon oluşturuyoruz. Bu fonksiyon receiver isimli bir parametre alıyor. Yukarıda kontratımızın sadece para çekmeye izin vereceğini söylemiştik. Buradaki receiver de işte bu alıcı oluyor. Bu alıcıya Algorand cüzdan adresini göndererek fonksiyonu çalıştıracağız. Şimdi fonksiyonun içine geçelim.

```
is_payment = Txn.type_enum() == TxnType.Payment
```

diyerek yapılan ödemenin türünü anlamaya çalışıyoruz. Burada Algorand tarafından sunulan bir tablo var. Bu tablo aşağıdaki şekilde.

TypeEnum mapping:

Index	"Type" string	Description
0	unknown	Unknown type. Invalid transaction.
1	pay	Payment
2	keyreg	KeyRegistration
3	acfg	AssetConfig
4	axfer	AssetTransfer
5	afrz	AssetFreeze



Biz burada `Txn.type_enum () == Int (1)` diyerek payment veya `Txn.type_enum () == Int (2)` diyerek KeyRegistration seçimi yapabiliyoruz. Benzer şekilde `AssetConfig`, `AssetTransfer`, `AssetFreeze` seçimi gerçekleştiriliyor. Fakat bunlar biraz daha ileri seviye bilgi içerdiği için şimdilik geçiyoruz.

`is_single_tx = Global.group_size() == Int(1)` bu yukarıdaki tabloda da izah edildiği gibi ödeme yapılacağı gösteriyor.

`is_correct_receiver = Txn.receiver() == Addr(receiver)` ile alıcı adresinin gerçek bir adres olup olmadığını kontrol ediyoruz.

`no_close_out_addr = Txn.close_remainder_to() == Global.zero_address()` ile global sıfır adrese ayarlandığını kontrol eder. Bu, işlemin adres bakiyesini kapatmayacağı anlamına gelir, işlemin alıcısını istenen alıcıya ayarlar ve işlem miktarını istenen tutara ayarlar. Yani bir tür adres ve işlem miktarı ayarlaması yapıyor.

`acceptable_fee = Txn.fee() <= Int(1000)` ile kabul edilen tutarı belli bir noktaya sınırlıyoruz. 1000 Algo olarak üst sınır koyuyoruz.

Ardından;

```
return And( is_payment,
is_single_tx,
is_correct_receiver,
no_close_out_addr,
no_rekey_addr,
acceptable_fee
)
```

diyerek yapılan her işlem için AND durumu yani True olması gerektiğini belirtiyoruz. Eğer bunlar doğru olursa para çekme işlemi sorunsuz gerçekleşecek.

Para çekme fonksiyonumuzu bu şekilde oluşturduk. Ama bu fonksiyon program açıldığında çalışan ana fonksiyon değil. Şimdi ana fonksiyonumuzu da oluşturalım.

```
if __name__ == "__main__":
    program = bank_for_account("ZZAF5ARA4MEC5PVDOP64J-
M5O5MQST63Q2KOY2FLYFLXXD3PFSNJJBFAFZM")
    print(compileTeal(program, Mode.Signature))
```

Aslında tek yaptığımız bizim fonksiyonumuzu çağırma işlemini belirlediğimiz cüzdanla yapmak. program olarak belirlediğimiz bu değişkeni de pyteal kütüphanesi içerisinde otomatik olarak gelen `compileTeal` fonksiyonuyla derlemek.



Kodumuz bu kadar. Artık bu kodu çalıştırdığımızda sözleşmemiz oluşacak. Çalıştıralım. Aşağıdaki şekilde bir sonuç döndü.

```
#pragma version 2
txn TypeEnum
int pay
==
global GroupSize
int 1
==
&&
txn Receiver
addr      ZZAF5ARA4MEC5PVDOP64JM5O5MQST63Q2KOY2FLYF-
LXXD3PFSNJJBIAFZM
==
&&
txn CloseRemainderTo
global ZeroAddress
==
&&
txn RekeyTo
global ZeroAddress
==
&&
txn Fee
int 1000
<=
&&
```

Dikkat ederseniz bu kod pragma ile başlayan bir akıllı kontrat. Bu sözleşmenin içinde ilk olarak pragma version 2 ile sözleşmenin versiyonu yazdırılıyor.

```
Ardından;
txn TypeEnum
int pay
global GroupSize
int 1
```

ile yukarıdaki tabloda verildiği şekilde bir ödeme yapılacağı gösteriliyor.



```
txn Receiver
addr ZZAF5ARA4MEC5PVDOP64JM5O5MQST63Q2KOY2FLYFLXX-
D3PFSNJJBAYFZM
```

alıcının adresini gösteriyor.

```
txn CloseRemainderTo
global ZeroAddress
==
&&
txn RekeyTo
global ZeroAddress
```

Buradaki ZeroAddressler, boş bir bayt dizisini temsil eden bir adrestir. Bir işlemde bir adres alanını boş bıraktığınızda kullanılır.

```
txn Fee
int 1000
<=
```

Koduyla da alınacak Algo için 1000 Algo üst limiti koyduk.

Artık kontratımız tamamen hazır. Python dilini kullanarak kontratı kodladık, pyteal compile işlemiyle bunu TEAL dilinde bir kontrata çevirdik.

Atomik Transferler

Bu bölümde Algo ile “Atomic Swap” yani Atomik Transfer işlemlerinin nasıl yapıldığını ele alacağız. Öncelikle, Atomic Swap nedir buna bakalım.

Geleneksel finasta, ticaret varlıkları genellikle her iki tarafın da kabul ettiklerini aldığından emin olmak için bir banka veya borsa gibi güvenilir bir aracı gerektirir. Bu banka veya borsa bir tür aracıdır. Algorand ekosisteminde bu araçları ortadan kaldırmak için Atomic Swap işlemleri kullanılır.

Bu basitçe, transferin parçası olan işlemlerin tamamının başarılı veya başarısız olduğunu kontrol eden kriptografik bir aracıdır. Atomik transferler, insanların güvenilir bir aracıya ihtiyaç duymadan varlık ticareti yapmalarına izin verirken, her bir tarafın da kabul ettiklerini alacağını garanti eder.

İşlemler bir mal veya hizmet karşılığında bir alıcıdan satıcıya Algos transferine izin vermektedir. Burada karşımıza bir kavram çıkıyor: “Hashed Time Locked Contract”



Bu yöntemde, alıcı ve satış fiyatı ile bir TEAL hesabına fon sağlar. Alıcı ayrıca bir gizli değer seçer ve TEAL programında bu değer güvenli bir hash'ini kodlar. Satıcı programdaki hash'e karşılık gelen gizli değeri sağlayabiliyorsa, TEAL programı bakiyesini satıcıya aktaracaktır. Satıcı mal veya hizmeti alıcıya teslim ettiğinde, alıcı programdaki secret'i açıklar. Satıcı secret'i anında doğrulayabilir ve ödemeyi geri çekebilir.

Atomik Transferler aşağıdaki gibi durumlarda kullanılabilir:

- Döngüsel ticaret - Ali, Veliye ancak Veli'nin Ayşe'ye ödeme yapması durumunda ödeme yapar. Yani eğer ikinci kişi üçüncü kişiye ödeme yapmazsa, birinci kişi de ikinci kişiye ödeme yapmaz.
- Grup ödemeleri - Herkesin ödemesi alınır veya hiç kimsenin alınmaz. Örneğin bu durum kitlesel fonlamalarda kullanılabilir.
- Merkezi olmayan borsalar - Merkezi bir borsadan geçmeden bir varlığın diğerleriyle ticaretinin yapılması.
- Dağıtılmış ödemeler - Birden çok alıcıya yapılan ödemeler.

Şimdi örnek kodumuza geçelim. Bu kodların güvenlik açısından denetlenmediğini, deneysel aşamada olduğu için güvenlik riskleri barındırabileceğini de not düşelim. Amacımız bu mekanizmayı öğrenmek.

from pyteal import * diyerek tüm pyteal kütüphanesini içe aktarıyoruz.

Blockzincir sistemlerinde bir teamül var. Satıcı için Alice, para alan kişi için de Bob takma ismi kullanılıyor. Bu teamüle uymak için kodumuzda bu isimleri kullanıyoruz.

İlk olarak bu iki kişinin hesap cüzdanlarını tanımlıyoruz.

```
alice = Addr("6ZHGGH5Z5CTPCF5WCESXMGRSVK7QJETR63M3N-
Y5FJCUYDHO57VTCTMJOBGY")
bob = Addr("7Z5PWO2C6LFNQFGHWKSK5H47IQP5OJW2M-
3HA2QPXTY3WTNP5NU2MHBW27M")
```

Ardından güvenlik kodu belirliyoruz.

```
secret = Bytes("base32", "232323232323")
```

timeout = 3000 diyerek işlem eğer 3 saniye içerisinde gerçekleşmezse iptal et diyoruz.

Şimdi bir fonksiyon tanımlayacağız. Önce fonksiyonun tamamını paylaşıp, ardından kodu yorumlayalım.



```
def htlc(tmpl_seller=alice,
        tmpl_buyer=bob,
        tmpl_fee=1000,
        tmpl_secret=secret,
        tmpl_hash_fn=Sha256,
        tmpl_timeout=timeout):
    fee_cond = Txn.fee() < Int(tmpl_fee)
    safety_cond = And(
        Txn.type_enum() == TxnType.Payment,
        Txn.close_remainder_to() == Global.zero_address(),
        Txn.rekey_to() == Global.zero_address(),
        ##global sıfır adrese ayarlandığını kontrol eder
        #(bu, işlemin adres bakiyesini kapatmayacağı anlamına gelir),
        #işlemin alıcısını istenen alıcıya ayarlar ve işlem miktarını istenen
        tutara
        # ayarlar
    )
    recv_cond = And(
        Txn.receiver() == tmpl_seller,
        tmpl_hash_fn(Arg(0)) == tmpl_secret
    )
    esc_cond = And(
        Txn.receiver() == tmpl_buyer,
        Txn.first_valid() > Int(tmpl_timeout)
    )
    return And(
        fee_cond,
        safety_cond,
        Or(recv_cond, esc_cond)
    )
tmpl_seller=alice,
tmpl_buyer=bob,
ile satıcı ve alıcı adreslerine atıfta bulunuyoruz.
tmpl_fee=1000,
tmpl_secret=secret,
tmpl_hash_fn=Sha256,
tmpl_timeout=timeout
```

değişkenleriyle. Ödenecek ücret limitini, gizlilik değerini, bu gizlilik değeri için nasıl bir şifreleme kullanılacağını ve timeout ile ne zaman zaman aşımı gerçekleşeceğini belirliyoruz.



Fonksiyona devam edelim.

`fee_cond = Txn.fee() < Int(tmp_fee)` ile bir tutar kontrolü yapıyoruz. Buradaki tutar 1000 Algo'dan düşük mü diye bakıyoruz. Burada da aslında şöyle bir amaç var. Kişiler yanlışlıkla yüksek tutar girebilir. Bir sıfır fazla girebilir. Blockchain'de geri dönüş olmadığı için bu durum bir krize yol açabilecektir. Sınır koyarak nispeten bunun önüne geçiyoruz.

```
safety_cond = And(  
    Txn.type_enum() == TxnType.Payment,  
    Txn.close_remainder_to() == Global.zero_address(),  
    Txn.rekey_to() == Global.zero_address(),  
)
```

Kısımında önceki uygulamamızda olduğu gibi bir AND yani, şu şu şartlar sağlansın diyoruz. Diyoruz ki buradaki enum tipi bir Payment yani ödeme olsun. Close remainder ve Rekey adresini de zero yani boş adrese eşitliyoruz. Bu sifıra eşitleme, işlemin adres bakiyesini kapatmayacağı anlamına gelir. Bunlar bizim güvenlik şartlarımızdı.

Şimdi alım koşullarını yazıyoruz.

```
recv_cond = And(  
    Txn.receiver() == tmp_seller,  
    tmp_hash_fn(Arg(0)) == tmp_secret  
)
```

Yukarıdaki kodda sadece alıcı fonksiyonuna satıcı bilgisini ve hash fonksiyonunun sıfıncı argümanı için secret bilgisini girdik. Alıcıya neden satıcı bilgisi girdik diye düşünebilirsiniz. Alıcıdan kasıt parayı alan kişi olduğundan bu kişi satıcı oluyor. Şimdi diğer güvenlik koşulumuzu tanımlayalım. Burada da tam tersi bir durum olacak.

```
esc_cond = And(  
    Txn.receiver() == tmp_buyer,  
    Txn.first_valid() > Int(tmp_timeout)  
)
```

Yukarıdaki koşulda da And işlemi için receiver yani mal veya hizmeti alan kişi, alıcıya eşitlenmiş. İlk durumda parayı alan satıcı var ikinci durumda ise mal veya hizmeti alan alıcı. Burayı karıştırmamak önemli. Ayrıca alıcı için bir timeout süresi var. Karşı taraf şifreyi veriyor ve bu tarafta bir doğrulama oluyor. Eğer 3 saniye içerisinde bu olmazsa işlem iptal oluyor. Bu kısım güvenlik açısından önemli. Algorand'ın atomik transferlerinde



vurguladığı “ya hep ya hiç” mottosunun arkasında da koddaki tam bu kısım var.

Şimdi ödeme, güvenlik, alıcı ve satıcı koşullarını yazdık. Alt durumlarının tamamının sağlanmasını istediğimizden bunların kendi içlerinde AND kapısı kullandık. Şimdi de bu dört koşulu AND kapısıyla birleştiriyoruz. Fakat önemli bir ayrıntı var. Ödeme ile güvenlik koşulları her durumda sağlanması gerektiği için doğrudan And içerisine yazıyoruz. Satıcı ve alıcı durumlarından ise yalnızca biri sağlanması gerektiği için bunlardan bir tanesini seçecek şekilde Or kapısı kullanarak ana And kapısı içine ekliyoruz.

```
return And(  
    fee_cond,  
    safety_cond,  
    Or(recv_cond, esc_cond)  
)
```

İşte bu kadar!

Bu bizim htlc isimli özel bir fonksiyonumuzdu. Uygulamamızın ana fonksiyonuna birkaç satır kod yazarak bu htlc fonksiyonu üzerinden akıllı kontratımızı oluşturalım.

```
if __name__ == "__main__":  
    print(compileTeal(htlc(), Mode.Signature))
```

Ana fonksiyon olan main fonksiyonunda compileTeal diyerek bir derleme işlemi yaptık. Parametre olarak htlc fonksiyonunu ve Mode.Signature ile imzamızı girdik. Kodumuzu çalıştırdığımızda TEAL dilinde akıllı kontratımız aşağıdaki şekilde oluşuyor.



```
#pragma version 2
txn Fee
int 1000
<
txn TypeEnum
int pay
==
txn CloseRemainderTo
global ZeroAddress
==
&&
txn RekeyTo
global ZeroAddress
==
&&
&&
txn Receiver
addr 6ZHGH5Z5CTPCF5WCESXMGRSVK7QJETR63M3NY5FJ-
CUYDHO57VTCMJOBGY
==
arg 0
sha256
byte base32(23232323232323)
==
&&
txn Receiver
addr 7Z5PWO2C6LFNQFGHWKSK5H47IQP5OJW2M3HA2QPXT-
Y3WTNP5NU2MHBW27M
==
txn FirstValid
int 3000
>
&&
||
&&
```

Bu sözleşmemizin içinde uygulamamızda kullandığımız; alıcı ve satıcı bilgileri, ödeme limiti, secret değeri, şifreleme türü gibi tüm değerler TEAL dilinde yazılmış durumda.



Yinelenen Takas Sözleşmesi

Akıllı kontratlar konusunda önemli bir başlık da, yinelenen takas sözleşmeleridir. Bu sözleşmeleri bir protokol gibi düşünebiliriz. İşlemi gerçekleştiren sağlayıcı tarafından yetkilendirme yapıldığı sürece belirlenen miktar (tmpl_amount) , belirlenen yere (tmpl_provider) tekrar tekrar gönderilebilir.

Daha da somutlaştırmak için örneklerle devam edelim. Diyelim ki bir sigortacı var ve bu takas sözleşmesini kurarak, belirlenen periyotta Algos transferi yapılmasını istiyor. Aynı şekilde bir hastaneyi düşünebiliriz. Hastane ilk işlemde first_valid imzasını kullanır ve sigorta sözleşmesi boyunca first_valid ile işlemini sunmaya devam eder. Algorand'ın altyapısının getirdiği kolaylık sayesinde bu işlemi sunarken not alanına da ek bilgiler girilebilir ve zaman aşımından sonra kalan bakiye geri alınabilir. Örnekler çoğaltılabilir. Şimdi yinelenen takas sözleşmesi konseptinde genel olarak aynı isimlendirmeye kullanılan değişkenleri ve özelliklerini inceleyelim.

Değişken	Anlamı
Tmpl_buyer	Hizmeti alan alıcı
Tmpl_provider	Hizmet sağlayıcı
Tmpl_ppk	Sağlayıcının genel anahtarı
Tmpl_amount	Hizmet başına ücretlendirilen microAlgo miktarı
Tmpl_fee	İzin verilen maksimum ücret miktarı
Tmpl_timeout	Alıcının Algo'larını geri alabileceği zaman aşım süresi

Bu değişkenleri son olarak örnek bir senaryoda yerleştirelim.

Ali (tmpl_buyer), Yaşam Hastanesi'nden (tmpl_provider) alacağı hizmet karşılığında aylık 200 TL bir ödeme (tmpl_amount) yapacak. Bunun için hastanenin 20a294b isimli genel anahtarını (tmpl_ppk) kullanarak ödemesini gerçekleştiriyor. Bu kontratta hastane tarafından hastalar için belirlenen maksimum ödeme tutarı 1000 TL (tmpl_fee). Bu ücret ödenirse tüm hizmetler alınabiliyor. Eğer Ali ödemesini yaptığında, ödeme karşı tarafa 3 dakika içerisinde geçmezse (tmpl_timeout) Algolar tekrar Ali'nin hesabına geri geliyor.

Şimdi kodlama kısmına geçelim. İlk olarak kodun tamamını paylaşıyoruz.



```

from pyteal import *

""" Recurring Swap
tmpl_buyer: hizmeti alan alıcı
tmpl_provider: hizmet sağlayıcısı
tmpl_ppk: sağlayıcının genel anahtarı
tmpl_amount: hizmet başına ücretlendirilen microAlgo miktarı
tmpl_fee: izin verilen maksimum işlem ücreti
tmpl_timeout: alıcının Algo'larını geri alabileceği zaman aşımı turu
"""

tmpl_buyer = Addr("6ZHGGH5Z5CTPCF5WCESXMGRSVK7QJETR63M-
3NY5FJCUYDHO57VTCMJOBGY")
tmpl_provider = Addr("7Z5PWO2C6LFNQFGHWKSK5H471QP5OJW2M-
3HA2QPXTY3WTNP5NU2MHBW27M")
tmpl_amount = Int(100000)
tmpl_fee = Int(1000)
tmpl_timeout = Int(100000)

def recurring_swap(tmpl_buyer=tmpl_buyer,
                  tmpl_provider=tmpl_provider,
                  tmpl_amount=tmpl_amount,
                  tmpl_fee=tmpl_fee,
                  tmpl_timeout=tmpl_timeout):
    fee_cond = Txn.fee() <= tmpl_fee
    type_cond = And(Txn.type_enum() == Int(1), Txn.rekey_to() == Global.
zero_address())
    rcv_cond = And(Txn.close_remainder_to() == Global.zero_address(),
Txn.receiver() == tmpl_provider,
Txn.amount() == tmpl_amount,
Ed25519Verify(Itob(Txn.first_valid()), Arg(0), tmpl_provider), #Itob,
uint64-> byte.
Txn.lease() == Sha256(Itob(Txn.first_valid()))

    close_cond = And(Txn.close_remainder_to() == tmpl_buyer,
Txn.amount() == Int(0),
Txn.first_valid() <= tmpl_timeout)

    program = And(fee_cond, type_cond, Or(rcv_cond, close_cond))

    return program

if __name__ == "__main__":
    print(compileTeal(recurring_swap(), Mode.Signature))

```




Şimdi adım adım analiz edelim. Değişkenler ile başlayalım. Aşağıdaki kısım sözel olarak ifade ettiğimiz Ali'nin hikâyesindeki; sırasıyla Ali'nin adresi, Hastane'nin adresi, ödenecek miktar, izin verilen maksimum tutar ve zaman aşımı.

```
tmpl_buyer = Addr("6ZHGHH5Z5CTPCF5WCESXMGRSVK7QJETR-
63M3NY5FJCUYDHO57VTCMJOBGY")
tmpl_provider = Addr("7Z5PWO2C6LFNQFGHWKSK5H47IQP5O-
JW2M3HA2QPXTY3WTNP5NU2MHBW27M")
tmpl_amount = Int(100000)
tmpl_fee = Int(1000)
tmpl_timeout = Int(100000)
```

Ardından `recurring_swap` ismi verilen ve yukarıda geçen tüm değişkenleri bir parametre olarak alan metodumuzu inceleyelim.

```
def recurring_swap(tmpl_buyer=tmpl_buyer,
    tmpl_provider=tmpl_provider,
    tmpl_amount=tmpl_amount,
    tmpl_fee=tmpl_fee,
    tmpl_timeout=tmpl_timeout):
    fee_cond = Txn.fee() <= tmpl_fee
    type_cond = And(Txn.type_enum() == Int(1), Txn.rekey_to() == Glo-
    bal.zero_address())
    recv_cond = And(Txn.close_remainder_to() == Global.zero_address(),
        Txn.receiver() == tmpl_provider,
        Txn.amount() == tmpl_amount,
        Ed25519Verify(Itob(Txn.first_valid()), Arg(0), tmpl_provider),
        #Itob, uint64-> byte.
        Txn.lease() == Sha256(Itob(Txn.first_valid()))))

    close_cond = And(Txn.close_remainder_to() == tmpl_buyer,
        Txn.amount() == Int(0),
        Txn.first_valid() <= tmpl_timeout)

    program = And(fee_cond, type_cond, Or(recv_cond, close_cond))

    return program
```



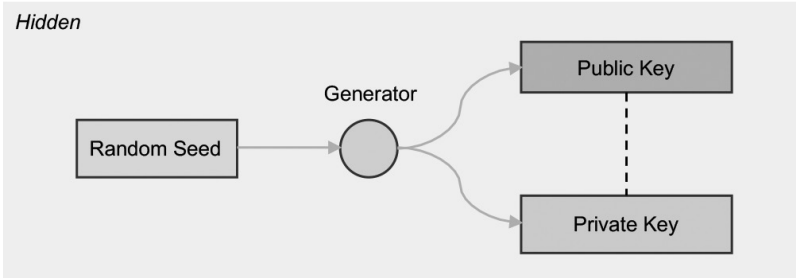
Kalan kodu tablo üzerinden satır satır inceleyelim.

fee_cond = Txn.fee() <= tmpl_fee	Yapılacak ödemenin, maksimum ödeme tutarından düşük olması gerektiğini vurguladık.
type_cond = And(Txn.type_enum() == Int(1),	Yapılacak işlemin bir ödeme olduğunu enum tiplerinden 1 olduğunu söyleyerek belirttik.
Txn.rekey_to() == Global.zero_address())	Rekey adresini boş adrese eşitledik.
recv_cond = And(Txn.close_remainder_to() == Global.zero_address(), Txn.receiver() == tmpl_provider, Txn.amount() == tmpl_amount, Ed25519Verify(Itob(Txn.first_valid()), Arg(0), tmpl_provider), #Itob, uint64-> byte. Txn.lease() == Sha256(Itob(Txn.first_valid())))	Bu bizim alıcının sağlayacağı durumlar kısmı. Kodun içinde değişkenleri eşliyoruz. Yani Txn.receiver fonksiyonunun, tmpl_provider değişkenini alacağını, miktar içinse Txn.amount() fonksiyonunun tmpl_amount değişkenini alacağını belirtiyoruz. Buralar rahatlıkla anlaşılabilir. Fakat kritik nokta Ed25519Verify ile başlayan kısım. Burada güvenlik için bir çalışma yapıyoruz. Detayına bu tablonun altında bakacağız.
close_cond = And(Txn.close_remainder_to() == tmpl_buyer, Txn.amount() == Int(0), Txn.first_valid() <= tmpl_timeout)	Bu kısım ise gönderici tarafından bazı koşulların sağlandığını gösteriyor. Remainder_to kısmı işlem bittikten sonra geri kalan kısmın tmpl_buyer da kalması yani alıcıda kalması ve yine bu noktada kalan miktarın 0'a eşit olması vurgulanıyor. Son olarak da zaman aşımının altında olması koşulu konuluyor.
program = And(fee_cond, type_cond, Or(recv_cond, close_cond))	Burada fee_cond, type_cond un kesinlikle sağlanması gerektiği, diğer iki koşuldan ise en az birinin sağlanması gerektiği söyleniyor. Yani ücret ve ödeme konusunda sabitlik var, fakat alıcı veya gönderici koşullarından biri sağlandığı takdirde işlem gerçekleşecek.
return program	Metodun çıktısının bir yukarıda program ile belirtilen And kapısına dönmesi söyleniyor.



Algorand'ın ifadesiyle buradaki Ed25519 imzalama yöntemi şu amaçla kullanılıyor:

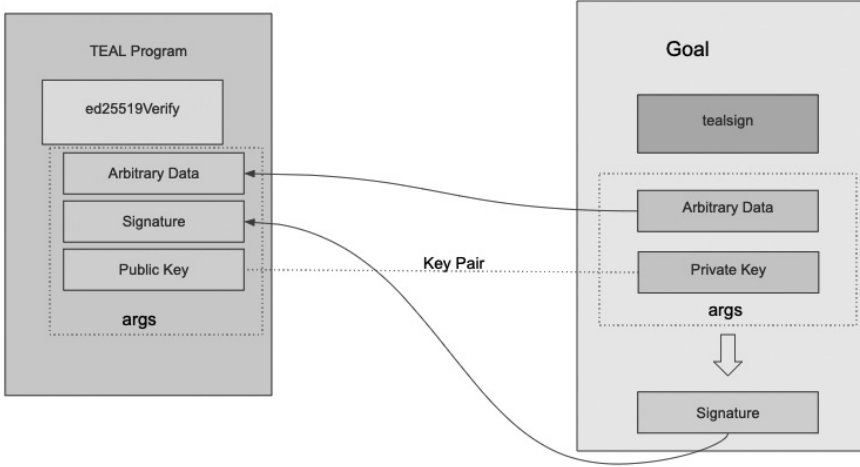
Algorand, Ed25519 yüksek hızlı, yüksek güvenli eliptik eğri imzaları kullanır. Anahtarlar, her bir SDK ile paketlenmiş standart, açık kaynaklı şifreleme kitaplıkları aracılığıyla üretilir. Anahtar üretme algoritması, girdi olarak rastgele bir değer alır ve bir genel anahtarı ve bununla ilişkili özel anahtarı temsil eden 32 baytlık iki dizi çıkarır. Bunlara ayrıca genel / özel anahtar çifti adı verilir. Bu anahtarlar, verilerin imzalanması ve imzaların doğrulanması gibi önemli kriptografik işlevleri yerine getirir. Bu anahtarlar insan tarafından okunabilir. Ayrıca aktarılırken insan hatasına karşı sağlam hale getirme ihtiyacını içeren nedenlerden dolayı, hem genel hem de özel anahtarlar dönüşüm geçirir. Bu dönüşümlerin çıktısı, geliştiricilerin çoğunluğunun ve genellikle tüm son kullanıcıların gördüğü şeydir. Aslında, Algorand geliştirici araçları, bu dönüşümlerde yer alan karmaşıklığı aktif olarak maskeleyemeye çalışır. Dolayısıyla, şifreleme ile ilgili kaynak kodunu değiştiren protokol düzeyinde bir geliştirici değilseniz, gerçek genel/özel anahtar çiftiyle asla karşılaşamayabilirsiniz.



Algorand'ın açıklamasında, geliştiriciyi buradaki kriptolojik detaylarla olabildiğince muhtatap etmek ismediğini öğreniyoruz. Zaten kodumuzda da dikkat ettiyseniz sabit bir kalıp kullanarak geçtik. Arka plandaki rastgele seçim, anahtarların üretimi ile ilgili bir şey yapmadık. Yani biz bu tüm kriptolojik süreç için sadece;

```
Ed25519Verify(Itob(Txn.first_valid()), Arg(0), tmpl_provider), #Itob, uint64-> byte.
```

Kodunu kullandık. Buradaki mimariye daha detaylı bakmak istersek aşağıdaki görsel bize yardımcı olacaktır.



(Ed25519 İmzalama Mekanizması)

Bizim akıllı kontratımızı hazır hale getirmek için bu kadar kriptoloji bilgisi bilmemiz yeterli. Daha fazla bilgi için bu imzayı da içine alan eliptik eğri imzaları konusunu araştırabilirsiniz.

Yine aynı kod bölümü içerisindeki `Txn.lease() == Sha256(Itob(Txn.first_valid()))` ile de bizim ilk belirlediğimiz tutar Sha256 şifrelemeyle kayıtlı ediliyor.

Yinelenen takas sözleşmesi için oluşturduğumuz `recurring_swap` fonksiyonu bu şekilde. Son olarak ana metodumuzdan bu metodu çağırıp `compileTeal` ile derleyip yazdırıyoruz.

```
if __name__ == "__main__":
    print(compileTeal(recurring_swap(), Mode.Signature))
```



Akıllı sözleşmemiz aşağıdaki şekilde oluşuyor.

```
#pragma version 2
txn Fee
int 1000
<=
txn TypeEnum
int 1
==
txn RekeyTo
global ZeroAddress
==
&&
&&
txn CloseRemainderTo
global ZeroAddress
==
txn Receiver
addr 7Z5PWO2C6LFNQFGHWKSK5H47IQP5OJW2M3HA2QPXT-
Y3WTNP5NU2MHBW27M
==
&&
txn Amount
int 100000
==
&&
txn FirstValid
itob
arg 0
addr 7Z5PWO2C6LFNQFGHWKSK5H47IQP5OJW2M3HA2QPXT-
Y3WTNP5NU2MHBW27M
ed25519verify
&&
txn Lease
txn FirstValid
itob
```



```

sha256
==
&&
txn CloseRemainderTo
addr 6ZHGHH5Z5CTPCF5WCESXMGRSVK7QJETR63M3NY5FJCUY-
DHO57VTCMJOBGY
==
txn Amount
int 0
==
&&
txn FirstValid
int 100000
<=
&&
||
&&

```

Paylaşımlı Ödeme

Blokzincir yapılarının bize sağladığı avantajlardan biri de bir ödemeyi gerekirse 1000 adrese bölerek gönderebilme özelliği. Diyelim ki bir afet durumunda afet durumundaki depremzedelere 10.000 TL bağış yapacaksınız fakat bunun 1000 farklı kişiye 10 TL olarak gittiğini garanti etmek istiyorsunuz. Geleneksel finans sisteminde bu bir hayli zahmetli bir iş olabilir. Fakat akıllı kontratlar ile bunu rahatlıkla yapabiliriz.

Benzer bir sistem şans oyunu tarzı sistemlerde de kullanılmaktadır. Yine içeriği görülebilen bir akıllı kontrat üzerinden yapılacak çekilişte toplam biriken tutar örneğin katılımcıların %90'ına bölünerek dağıtılabilir. Fakat şans oyunu tarzı uygulamalar yasal olarak farklı bir statüde ele alındığı için hukuksal boyutuna hakim olmadan böyle bir Dapps yani, dağıtık uygulama geliştirmeniz cezai süreçlere yol açabilmektedir.

Biz örneğimizde hesap sahibinin ödemesini önceden belirlediğimiz 2 adrese belirlediğimiz oranda bölerek göndereceğiz. Eğitim amaçlı 2 adres kullanıldı. 2 değil 20 de 200 de olabilir.

Öncelikle kodumuzun tamamını paylaşalım. Ardından bölüm bölüm yorumlayalım.



```
from pyteal import *

#Split Payment, tmpl_own tarafından tmpl_rcv1 ve tmpl_rcv2 arasın-
daki ödemeyi tmpl_ratn / tmpl_ratd oranında
# bölerek gönderim yapar.

""""Split Payment""""

tmpl_fee = Int(1000)
tmpl_rcv1 = Addr("6ZHGH5Z5CTPCF5WCESXMGRSVK7QJETR-
63M3NY5FJCUYDHO57VTCMJOBGY")
tmpl_rcv2 = Addr("7Z5PWO2C6LFNQFGHWKSK5H47IQP5OJW2M-
3HA2QPXTY3WTNP5NU2MHBW27M")
tmpl_own = Addr("5MK5NGBRT5RL6IGUSYDIX5P7TNNZKRVXKT-
6FGVI6UVK6IZAWTYQGE4RZIQ")
tmpl_ratn = Int(1)
tmpl_ratd = Int(3)
tmpl_min_pay = Int(1000)
tmpl_timeout = Int(3000)

def split(tmpl_fee=tmpl_fee,
          tmpl_rcv1=tmpl_rcv1,
          tmpl_rcv2=tmpl_rcv2,
          tmpl_own=tmpl_own,
          tmpl_ratn=tmpl_ratn,
          tmpl_ratd=tmpl_ratd,
          tmpl_min_pay=tmpl_min_pay,
          tmpl_timeout=tmpl_timeout):
    split_core = And(
        Txn.type_enum() == TxnType.Payment,
        Txn.fee() < tmpl_fee,
        Txn.rekey_to() == Global.zero_address()
    )

    split_transfer = And(
        Gtxn[0].sender() == Gtxn[1].sender(),
        Txn.close_remainder_to() == Global.zero_address(),
        Gtxn[0].receiver() == tmpl_rcv1,
        Gtxn[1].receiver() == tmpl_rcv2,
```



```

    Gtxn[0].amount() == ((Gtxn[0].amount() + Gtxn[1].amount()) *
    tmp1_ratn) / tmp1_ratd,
    Gtxn[0].amount() == tmp1_min_pay
)

split_close = And(
    Txn.close_remainder_to() == tmp1_own,
    Txn.receiver() == Global.zero_address(),
    Txn.amount() == Int(0),
    Txn.first_valid() > tmp1_timeout
)

split_program = And(
    split_core,
    If(Global.group_size() == Int(2),
        split_transfer,
        split_close
    )
)

return split_program

if __name__ == "__main__":
    print(compileTeal(split(), Mode.Signature))

```

Yine ilk satırda pyteal kütüphanemizi içe aktararak başlıyor ve ardından da değişkenlerimizi tanımlıyoruz.

Değişkenleri inceleyerek başlayalım.

```

tmp1_fee = Int(1000)
tmp1_rcv1 = Addr("6ZHGGHH5Z5CTPCF5WCESXMGRSVK7QJETR-
63M3NY5FJCUYDHO57VTCMJOBGY")
tmp1_rcv2 = Addr("7Z5PWO2C6LFNQFGHWKSK5H47IQP5OJW2M-
3HA2QPXTY3WTNP5NU2MHBW27M")
tmp1_own = Addr("5MK5NGBRT5RL6IGUSYDIX5P7TNNZKRVXKT-
6FGVI6UVK6IZAWTYQGE4RZIQ")
tmp1_ratn = Int(1)
tmp1_ratd = Int(3)
tmp1_min_pay = Int(1000)
tmp1_timeout = Int(3000)

```




Değişken	Anlamı
Tmpl_fee	Ödemelerdeki maksimum sınır. 1000 Algo olarak belirledik.
tmpl_rcv1	Para taksimat yapıldıktan sonra gönderilecek ilk adres.
tmpl_rcv2	Para taksimat yapıldıktan sonra gönderilecek ikinci adres.
tmpl_own	Parayı gönderen kişinin adresi.
tmpl_ratn	Taksimat sırasında ilk adrese yapılacak ödemenin payı. (1)
tmpl_ratd	Taksimat sırasında ikinci adrese yapılacak ödemenin payı. (3)
tmpl_min_pay	Minumum ödeme tutarı. 1000 Algo olarak belirledik.
tmpl_timeout	İşlemin gerçekleşmesi için geçen zaman aşımı. Minisaniye türünden giriliyor. 3000 girişi 3 saniyeye karşılık geliyor.

Değişkenlerimizi tanımladıktan sonra split isimli metodumuzu oluşturacağız. En sonda da ana metoddan split metodunu çağıracağız.

Metodumuzu oluştururken yukarıda kullandığımız her değişkeni bir parametre olarak belirliyoruz.

```
def split(tmpl_fee=tmpl_fee,
          tmpl_rcv1=tmpl_rcv1,
          tmpl_rcv2=tmpl_rcv2,
          tmpl_own=tmpl_own,
          tmpl_ratn=tmpl_ratn,
          tmpl_ratd=tmpl_ratd,
          tmpl_min_pay=tmpl_min_pay,
          tmpl_timeout=tmpl_timeout):
```

Ardından da metodumuzun içine koşulları girmeye başlıyoruz. İlk koşulumuz split_core isminde.



```
split_core = And(
    Txn.type_enum() == TxnType.Payment,
    Txn.fee() < tmpl_fee,
    Txn.rekey_to() == Global.zero_address()
)
```

Burada bir AND kapısı kullanarak; işlemin bir ödeme olduğunu TxnType.Payment ile, ödemenin maksimum ödeme tutarını aşmayacağını tmpl_fee ile ve Txn.rekey_to adresinin boş bir değer olduğunu Global.zero_address() ile belirliyoruz.

İkinci koşulumuza geçelim. Bu koşulumuzun ismi de split_transfer.

Burada kod biraz karışık gelebilir önce kısmı paylaşıp ardından dikkatlice inceleyelim.

```
split_transfer = And(
    Gtxn[0].sender() == Gtxn[1].sender(),
    Txn.close_remainder_to() == Global.zero_address(),
    Gtxn[0].receiver() == tmpl_rcv1,
    Gtxn[1].receiver() == tmpl_rcv2,
    Gtxn[0].amount() == ((Gtxn[0].amount() + Gtxn[1].amount()) *
    tmpl_ratn) / tmpl_ratd,
    Gtxn[0].amount() == tmpl_min_pay
)
```

Gtxn[0].sender() == Gtxn[1].sender(), yani [0] numaralı hesabın göndericisiyle [1] numaralı hesabın göndericisi aynı kişi. Bu da bizim ilk başta tmpl_own değişkeniyle tanımladığımız hesap olacak.

Txn.close_remainder_to() == Global.zero_address(), işlem gerçekleştikten sonra adresi zero adrese eşitle. Bu işlem sonrası zero adrese eşitleme artık bir varsayılan gibi akıllı kontratlarda.

Gtxn[0].receiver() == tmpl_rcv1,

Gtxn[1].receiver() == tmpl_rcv2, satırlarıyla birinci ve ikinci alıcının kim olduğunu belirtiyoruz.

Gtxn[0].amount() == ((Gtxn[0].amount() + Gtxn[1].amount()) * tmpl_ratn) / tmpl_ratd, bu kısım tamamen matematik. [0] numaralı alıcıya gidecek para [0] ve [1] in toplamının tmpl_ratn ile çarpımı ve bu sonucun da tmpl_ratd e bölümü. Burada bir oran orantı işlemi yapıyoruz. Siz kendi kontratınızda bu değerler ile oynayabilirsiniz.

Gtxn[0].amount() == tmpl_min_pay, kısmında ise [0] numaralı yani daha az para alacak hesabın ücreti minumum ödeme tutarına eşit olsun diyoruz.



Split_Transfer koşulu içinde yer alan bu durumların tamamını AND kapısından geçirerek her birinin doğrulanması gerektiğini belirtiyoruz.

Sonraki koşulumuzun ismi split_close. Kodumuz şu şekilde.

```
split_close = And(  
    Txn.close_remainder_to() == tmpl_own,  
    Txn.receiver() == Global.zero_address(),  
    Txn.amount() == Int(0),  
    Txn.first_valid() < tmpl_timeout  
)
```

Txn.close_remainder_to() == tmpl_own, kalan para hesabın sahibinde kalsın. Alıcının adresi Txn.receiver() == Global.zero_address(), ile sıfırlansın. Tutar Txn.amount() == Int(0), ile sıfırlansın. Txn.first_valid() > tmpl_timeout kısmıyla zaman aşımının altında bir sürede işlemin gerçekleştiğini belirtiyoruz. Yine bu koşulların hepsini AND kapısından geçirerek koşulumuzu oluşturuyoruz.

Ardından split_program ile ana koşulumuzu oluşturup önceki koşulları topluca ele alıyoruz.

```
split_program = And(  
    split_core,  
    If(Global.group_size() == Int(2),  
        split_transfer,  
        split_close  
    )  
)
```

Bu ana kapıda split_core u direk alıyor. Sonrasında grup boyutu 2 ise, yani iki kişi ödeme alıyorsa, split_transfer ve split_close u da sağlasın diyoruz.

Son olarak; return split_program metodumuzun split_program koşuluna dönmesini istiyoruz.

Metodumuz tamamen hazır. Sadece ana metoddan çağırarak derleme işlemini yapmamız gerekiyor. Bunun için de aşağıdaki kodu kullanıyoruz. Pyteal kütüphanesi içindeki compileTeal hazır fonksiyonuyla Python kodumuzu TEAL dilinde bir akıllı kontrata çeviriyoruz.



```
if __name__ == "__main__":  
    print(compileTeal(split(), Mode.Signature))
```

Çıktıya bakalım.

```
#pragma version 2  
txn TypeEnum  
int pay  
==  
txn Fee  
int 1000  
<  
&&  
txn RekeyTo  
global ZeroAddress  
==  
&&  
global GroupSize  
int 2  
==  
bnz l0  
txn CloseRemainderTo  
addr 5MK5NGBRT5RL6IGUSYDIX5P7TNNZKRVXKT6FGVI6UVK6I-  
ZAWTYQGE4RZIQ  
==  
txn Receiver  
global ZeroAddress  
==  
&&  
txn Amount  
int 0  
==
```



```

&&
txn FirstValid
int 3000
>
&&
b 11
l0:
gtxn 0 Sender
gtxn 1 Sender
==
txn CloseRemainderTo
global ZeroAddress
==
&&
gtxn 0 Receiver
addr 6ZHGHH5Z5CTPCF5WCESXMGRSVK7QJETR63M3NY5FJCUY-
DHO57VTCMJOBGY
==
&&
gtxn 1 Receiver
addr 7Z5PWO2C6LFNQFGHWKSK5H47IQP5OJW2M3HA2QPXT-
Y3WTNP5NU2MHBW27M
==
&&
gtxn 0 Amount
gtxn 0 Amount
gtxn 1 Amount
+
int 1
*
int 3
/
==
&&
gtxn 0 Amount
int 1000
==
&&
l1:
&&

```

Yukarıdaki şekilde kontratımız sorunsuz bir şekilde oluştu.



Periyodik Ödemeler

Akıllı kontratlar konusunu araştırırken ilk dikkat çekici şeylerden biri mevcut finansal sistemimiz ve onun merkezinde oluşan dünyamızın belli bir hızda olduğuydu. Blokzincirin ortaya çıkaracağı ekosistemde ise bu hız yeterli olmayacak ve çok daha ötesine geçilecekti.

Örneğin para birimi olarak TL, Dolar gibi birimler kullanıyoruz. Bunların alt kırımında ise kuruş, cent var. Peki, hiç düşündünüz mü? Kuruşun veya centin 100'de 1'i veya 1000'de 1'i için bir tanımlama neden yok? Çünkü mevcut finansal sistemde bu hız yeterli. Oysa ki Bitcoin'de virgülden sonra 6 hane var. En küçük birime Satoshi ismi veriliyor. Burada odaklanmamız gereken nokta küsüratlar üzerinden kurulan sistemlerin ne kadar esnek olabileceği.

Somut bir örnek verelim. Bugün üniversite öğrencilerinin yüzde kaç burs alabiliyor? Çok düşük bir yüzdesi. Peki neden? Çünkü burs sistemi geleneksel finans sistemi üzerinden hantal yürüyor. Bir kişiye 300 TL bursu başka bir kişinin direk vermesi zor. Fakat bu 300 TL, 300 farklı kişiden 1'er TL olarak kesilebilir. Gayet güzel. Peki, bunu bankacılık sistemiyle, otomatik ödeme talimatlarıyla yapmak ne kadar kolay? Bir de arada aracılar olacak? Bunun yerine aradaki aracı kriptoloji olsa ve kesinti olmadan ödemeler yapılırsa. Akıllı kontrat üzerinden yapılan bu ödemenin kaynak kodları görülebileceği için mevcut burs sistemlerine göre çok daha şeffaf olarak gerçekleşebilecek. 7/24 gerçekleşebilecek ve hatta "abartıyorum" 1 kuruş burs vermek isteyen kişi dahi burs verebilecek.

Burs örneğinde biraz somutlaştırmaya çalıştığımız periyodik ödemelerin gelecekte nasıl değişebileceğiydi. Benzer durumları maaşlar için de düşünebiliriz. Şu anda maaşlar aylık veriliyor. Haftalık veya günlük ödemeler de yapılabilir. Fakat gelecekte, belki giyilebilir bir teknoloji veya farklı bir teknoloji sayesinde, değil saatlik dakikalık bile hesaplamalar yapıp ödeme yapılabilir. Bu da bir kişinin dakikalarını dolu geçirmesin, birkaç farklı işte çalışabilmesine yol açarken iş hayatının kodları değişecek.

Az sonra örneğini vereceğimiz akıllı kontratı bu önbilgilerle düşünmekte fayda var.

Öncelikle kodumuzun tamamını paylaşalım ardından da yorumlayalım.



Bu örnek yalnızca bilgi amaçlı verilmiştir ve güvenlik açısından denetlenmemiştir.

```
from pyteal import *
```

```
“““Periyodik Ödeme “““
```

```
tmpl_fee = Int(1000)
```

```
tmpl_period = Int(50)
```

```
tmpl_dur = Int(5000)
```

```
tmpl_lease = Bytes("base64", "023sdDE2")
```

```
tmpl_amt = Int(2000)
```

```
tmpl_rcv = Addr("6ZHGHH5Z5CTPCF5WCESXMGRSVK7QJETR-63M3NY5FJCUYDHO57VTCMJOBGY")
```

```
tmpl_timeout = Int(30000)
```

```
def periodic_payment(tmpl_fee=tmpl_fee,
    tmpl_period=tmpl_period,
    tmpl_dur=tmpl_dur,
    tmpl_lease=tmpl_lease,
    tmpl_amt=tmpl_amt,
    tmpl_rcv=tmpl_rcv,
    tmpl_timeout=tmpl_timeout):
```

```
    periodic_pay_core = And(
        Txn.type_enum() == TxnType.Payment,
        Txn.fee() < tmpl_fee,
        Txn.first_valid() % tmpl_period == Int(0),
        Txn.last_valid() == tmpl_dur + Txn.first_valid(),
        Txn.lease() == tmpl_lease
    )
```

```
    periodic_pay_transfer = And(
        Txn.close_remainder_to() == Global.zero_address(),
        Txn.rekey_to() == Global.zero_address(),
        Txn.receiver() == tmpl_rcv,
        Txn.amount() == tmpl_amt
    )
```

```
    periodic_pay_close = And(
        Txn.close_remainder_to() == tmpl_rcv,
        Txn.rekey_to() == Global.zero_address(),
        Txn.receiver() == Global.zero_address(),
        Txn.first_valid() == tmpl_timeout,
        Txn.amount() == Int(0)
    )
```



```
periodic_pay_escrow = periodic_pay_core.And(periodic_pay_trans
fer.Or(periodic_pay_close))
return periodic_pay_escrow
if __name__ == "__main__":
print(compileTeal(periodic_payment(), Mode.Signature))
```

Kodumuzda Pyteal kütüphanesini kullanarak otomatik Algo ödemesi yapan bir akıllı kontrat yazdık. Diğer örneklerimizde olduğu gibi ilk olarak kütüphaneyi içe aktardık, ardından bir metod oluşturup, main metodumuzdan bu önceden oluşturduğumuz metodu çağırdık.

Değişkenlerle başlıyoruz:

```
tmpl_fee = Int(1000)
tmpl_period = Int(50)
tmpl_dur = Int(5000)

tmpl_lease = Bytes("base64", "023sdDE2")
tmpl_amt = Int(2000)
tmpl_rcv = Addr("6ZHGH5Z5CTPCF5WCESXMGRSVK7QJETR-
63M3NY5FJCUYDHO57VTCMJOBGY")
tmpl_timeout = Int(30000)
```

Değişken	Anlamı
Tmpl_fee	Maksimum ödeme limiti. Yanlış ödemelerden koruyor.
Tmpl_Period	Periyod, 50 günde bir yap
Tmpl_Lease	İşlemi gerçekleştir. Şifreleme ve gizli anahtar kullanımı burada bulunuyor.
Tmpl_Dur	Süre, 5000 saniyede bir yap.
Tmpl_Amt	Miktar, ödeme yapılacak miktar.
Tmpl_Rcv	Alıcı, ödemeyi alacak kişi.
Tmpl_Time_out	İşlem gerçekleşmezse, zaman aşım süresi.

Metodumuzu oluşturalım. İsmi periodic_payment oluyor ve yukarıdaki



değişkenlerin tamamını parametre olarak alıyoruz. Bu aslında detaylı bir periyodik ödeme kontratı. Her zaman bu kadar detaya gerek yok fakat bakış açımızın gelişmesi için bu örneği ele aldık.

Şimdi metodumuzun içindeki en kritik noktaya odaklanalım. 3 tane farklı sorgulama işlemi yapıyoruz.

İlk sorgulama: `periodic_pay_core`

```
periodic_pay_core = And(  
    Txn.type_enum() == TxnType.Payment,  
    Txn.fee() < tmpl_fee,  
    Txn.first_valid() % tmpl_period == Int(0),  
    Txn.last_valid() == tmpl_dur + Txn.first_valid(),  
    Txn.lease() == tmpl_lease  
)
```

Burada `TxnType.Payment` ile ödemenin tipini belirtiyoruz. Bu bir ödeme dir diyoruz. Ardından maksimum ödeme limitini geçip geçmediğini `Txn.fee() < tmpl_fee` ile kontrol ediyoruz. Bir sonraki satırda `Txn.first_valid() % tmpl_period == Int(0)`, ile bir mod alma işlemi yapıyoruz. Çünkü periyodumuzun dolduğu tarihte bir sıfırlama gerekiyor. Yani 30 gün periyodlu bir ödeme için 35. günün modu 30'a göre alınrsa 5 olur ve ödeme kabul edilmez. Ama 60. Gün için mod 30'a göre kalan 0 olur ve bu eşitlik sağlanır. Ödeme kabul edilir. Altındaki ise sonraki ödemeler kullanılıyor. Burada da ilk doğrulamanın üstüne `tmpl_dur` koyarak bir sonraki doğrulamayı buluyor. Son satırda da `tmpl_lease` ile base64 şifrelemede belirlenen şifreyle ödemeyi gerçekleştiriyor. Burada özetle ödemenin periyodu ve zamanı gelince yapılacağını söyledik.

İkinci sorgulama: `periodic_pay_transfer`

```
periodic_pay_transfer = And(  
    Txn.close_remainder_to() == Global.zero_address(),  
    Txn.rekey_to() == Global.zero_address(),  
    Txn.receiver() == tmpl_rcv,  
    Txn.amount() == tmpl_amt  
)
```

İkinci sorgulamamızda ödemenin tam gerçekleştiği ana odaklanıyoruz. Burada bir alıcı bir de verici tarafı var. `Global.zero_address()` kısımları ilk uygulamalardan aşına olduğumuz bir yapı. Alıcının bu değerlerini gizlilik-



ten ötürü bilmiyoruz ve bu yüzden boş değer anlamına gelen zero adres kullanıyoruz. Geriye kalan `tmpl_rcv` ve `tmpl_amt` değişkenlerini de ödemenin yapılacağı adres ve miktarı doğrulamakta kullanıyoruz. İkinci sorgulamada aslında alıcı taraftaki değerleri giriyoruz.

Üçüncü sorgulama: `periodic_pay_close`

Bu sorgulamada ise ödemeyi gönderen, yapan kişi tarafındaki değerleri giriyoruz.

```
periodic_pay_close = And(
  Txn.close_remainder_to() == tmpl_rcv,
  Txn.rekey_to() == Global.zero_address(),
  Txn.receiver() == Global.zero_address(),
  Txn.first_valid() == tmpl_timeout,
  Txn.amount() == Int(0)
)
```

`Txn.close_remainder_to()` ile kendinde ne kadar kaldığını söylüyor. `Global.zero_address()` değerleri kullanılıyor. Ekstra olarak da timeout ile ödemenin geçip geçmediği kontrol ediliyor. Belirlenen sürede olmazsa ödeme gerçekleşmiyor. `Txn.amount` değerini 0'a eşitleyerek ödemenin gerçekleştiği teyit ediliyor.

Özetlersek, ilk sorgulamada ödemenin zamanının gelip gelmedi, ikinci sorgulamada alıcı tarafından teyit, üçüncü sorgulamada ise parayı gönderen tarafından teyit değerlerini girdik.

Biz bu üç sorgulamayı birleştirirken ilk sorgulamayı kesinlikle alıyoruz. Dolayısıyla onu AND içerisinde alıp ikinci veya üçüncü sorgulamalardan birini alıyoruz. Burada da OR kapısı kullanıyoruz.

Bundan dolayı da kodumuz aşağıdaki şekilde oluyor.

```
periodic_pay_escrow = periodic_pay_core.And(periodic_pay_transfer.
  Or(periodic_pay_close))
return periodic_pay_escrow
```

Son olarak kodumuzu compile ederek TEAL diline çeviriyoruz. Sözleşmemiz aşağıdaki şekilde sorunsuz olarak oluştu.



```
#pragma version 2
txn TypeEnum
int pay
==
txn Fee
int 1000
<
&&
txn FirstValid
int 50
%
int 0
==
&&
txn LastValid
int 5000
txn FirstValid
+
==
&&
txn Lease
byte base64(023sdDE2)
==
&&
txn CloseRemainderTo
global ZeroAddress
==
txn RekeyTo
global ZeroAddress
==
&&
txn Receiver
addr 6ZHGH5Z5CTPCF5WCESXMGRSVK7QJETR63M3NY5FJCUY-
DHO57VTCMJOBGY
==
&&
txn Amount
int 2000
==
&&
```



```

txn CloseRemainderTo
addr 6ZHGHH5Z5CTPCF5WCESXMGRSVK7QJETR63M3NY5FJCUY-
DHO57VTCMJOBGY
==
txn RekeyTo
global ZeroAddress
==
&&
txn Receiver
global ZeroAddress
==
&&
txn FirstValid
int 30000
==
&&
txn Amount
int 0
==
&&
||
&&

```

Bu örnekte belirlediğimiz adrese Algo türünden belirlenen ölçütlere göre ödeme yapan bir akıllı kontrat yapmış olduk.

Oylama Sistemi

Oylama, hesapların rastgele seçimler için kaydolmasına ve oy kullanmasına izin verir. Burada seçim (choice) herhangi bir bayt dilimidir ve herkesin oy kullanmak için kaydolmasına izin verilir. Bu örnekte, RegBegin ve RegEnd global durumu tarafından tanımlanan, hesapların ne zaman oy kullanmak için kaydolabileceğini kısıtlayan yapılandırılabilir bir oylama dönemi (voting period) vardır. Ayrıca, oylamanın ne zaman gerçekleşebileceğini kısıtlayan, global durum VotingBegin ve VotingEnd tarafından tanımlanan ayrı bir yapılandırılabilir oylama dönemi vardır.

Oy verebilmek için bir hesabın kayıt olması gerekir. Hesaplar birden fazla oy kullanamaz ve bir hesap, oylama süresi bitmeden uygulamadan çekilirse, oyları iptal edilir. Sonuçlar, başvurunun global durumunda görülebilir ve kazanan, en fazla oyu alan adaydır.

Biz de bir oylama uygulaması geliştireceğiz.



İlk olarak programımızı paylaşalım ardından kodu adım adım inceleyelim.

```
from pyteal import *

def approval_program():
    on_creation = Seq([
        App.globalPut(Bytes("Creator"), Txn.sender()),
        Assert(Txn.application_args.length() == Int(4)),
        App.globalPut(Bytes("RegBegin"), Btoi(Txn.application_args[0])),
        #Btoi: Bir bayt dizesini uint64'e dönüştürme
        App.globalPut(Bytes("RegEnd"), Btoi(Txn.application_args[1])),
        App.globalPut(Bytes("VoteBegin"), Btoi(Txn.application_args[2])),
        App.globalPut(Bytes("VoteEnd"), Btoi(Txn.application_args[3])),
        Return(Int(1))
    ])

    is_creator = Txn.sender() == App.globalGet(Bytes("Creator"))

    get_vote_of_sender = App.localGetEx(Int(0), App.id(), Bytes("voted"))

    on_closeout = Seq([
        get_vote_of_sender,
        If(And(Global.round() <= App.globalGet(Bytes("VoteEnd")), get_vote_of_sender.hasValue()),
            App.globalPut(get_vote_of_sender.value(), App.globalGet(get_vote_of_sender.value()) - Int(1))
        ),
        Return(Int(1))
    ])

    on_register = Return(And(
        Global.round() >= App.globalGet(Bytes("RegBegin")),
        Global.round() <= App.globalGet(Bytes("RegEnd"))
    ))

    choice = Txn.application_args[1]
    choice_tally = App.globalGet(choice)
    on_vote = Seq([
```



```

Assert(And(
    Global.round() >= App.globalGet(Bytes("VoteBegin")),
    Global.round() <= App.globalGet(Bytes("VoteEnd"))
)),
get_vote_of_sender,
If(get_vote_of_sender.hasValue(),
    Return(Int(0))
),
App.globalPut(choice, choice_tally + Int(1)),
App.localPut(Int(0), Bytes("voted"), choice),
Return(Int(1))
])

program = Cond(
    [Txn.application_id() == Int(0), on_creation],
    [Txn.on_completion() == OnComplete.DeleteApplication, Return(
is_creator)],
    [Txn.on_completion() == OnComplete.UpdateApplication, Return(
is_creator)],
    [Txn.on_completion() == OnComplete.CloseOut, on_closeout],
    [Txn.on_completion() == OnComplete.OptIn, on_register],
    [Txn.application_args[0] == Bytes("vote"), on_vote]
)

return program

def clear_state_program():
    get_vote_of_sender = App.localGetEx(Int(0), App.id(), Bytes("vo-
ted"))
    program = Seq([
        get_vote_of_sender,
        If(And(Global.round() <= App.globalGet(Bytes("VoteEnd")), get_
vote_of_sender.hasValue()),
            App.globalPut(get_vote_of_sender.value(), App.globalGet(get_
vote_of_sender.value()) - Int(1))
        ),
        Return(Int(1))
    ])

```



```
return program

if __name__ == "__main__":
    with open('vote_approval.teal', 'w') as f:
        compiled = compileTeal(approval_program(), Mode.Application)
        f.write(compiled)

    with open('vote_clear_state.teal', 'w') as f:
        compiled = compileTeal(clear_state_program(), Mode.Application)
        f.write(compiled)
```

İlk olarak main metodumuzun içindeki bir noktaya dikkat çekerek başlayalım. Önceki örneklerimizde oluşturduğumuz bir metodu, main metodu içinden çağırıyorduk ve TEAL diline derleme yapıyorduk. Burada ise durum biraz daha kompleks.

```
if __name__ == "__main__":
    with open('vote_approval.teal', 'w') as f:
        compiled = compileTeal(approval_program(), Mode.Application)
        f.write(compiled)

    with open('vote_clear_state.teal', 'w') as f:
        compiled = compileTeal(clear_state_program(), Mode.Application)
        f.write(compiled)
```

Main metodumuzun içinde iki farklı TEAL dosyası var. Bunlardan biri vote_approval diğeriye vote_clear_state dosyaları. Tıpkı diğer uygulamalarda olduğu gibi compileTeal fonksiyonuyla approval_program ve clear_state_program dosyalarını derliyoruz. Fakat buradaki ekstra durum şu. Derleme sonucu oluşan akıllı kontratları yukarıda bahsi geçen 2 TEAL dosyasının içine yazıyoruz. Demek ki biz iki metod üzerinde çalışacağız.

Şimdi bu uygulamada ilk kez göreceğimiz bazı kullanımlara yakından bakalım. Periyodik ödeme gibi önceki uygulamalarda AND ve OR kapılarını kullanmak yeterli oluyordu. Bu örneğimizde ise Cond isimli bir kapı var.

Bu kapı bir if else gibi çalışıyor fakat farklı bir yazımı var. Öncelikle koddaki ilgili kısmı alalım.



```

program = Cond(
  [Txn.application_id() == Int(0), on_creation],
  [Txn.on_completion() == OnComplete.DeleteApplication, Return(is_creator)],
  [Txn.on_completion() == OnComplete.UpdateApplication, Return(is_creator)],
  [Txn.on_completion() == OnComplete.CloseOut, on_closeout],
  [Txn.on_completion() == OnComplete.OptIn, on_register],
  [Txn.application_args[0] == Bytes("vote"), on_vote]
)

```

Burada Cond ifadesinin içinde yer alan virgüllü yapı bize şunu söylüyor. Eğer;

Bu durum olursa	Şunu yap
<code>Txn.application_id() == Int(0)</code>	<code>on_creation</code>
<code>Txn.on_completion() == OnComplete.DeleteApplication</code>	<code>Return(is_creator</code>
...	...

Bu şekilde gidiyor. Yani virgülün solundaki durum olursa, sağdaki işlemi yap. Eğer `application_id` 0 olursa `on_creation` u çağır gibi. Bu bizim akıllı kontratlarda kullanacağımız Cond yapısı veya bir başka ifadeyle Cond kapısı.

İkinci bir yapı ise Seq yapısı. Seq , sequence yani dizi kelimesinin kısaltması. İşlevi de verilen işlemleri bir diziye koyarak sırayla gerçekleştirmek. Kodumuzun ilgili kısmını alalım.

```

on_creation = Seq([
  App.globalPut(Bytes("Creator"), Txn.sender()),
  Assert(Txn.application_args.length() == Int(4)),
  App.globalPut(Bytes("RegBegin"), Btoi(Txn.application_args[0])),
  #Btoi: Bir byte dizesini uint64'e dönüştürme
  App.globalPut(Bytes("RegEnd"), Btoi(Txn.application_args[1])),
  App.globalPut(Bytes("VoteBegin"), Btoi(Txn.application_args[2])),
  App.globalPut(Bytes("VoteEnd"), Btoi(Txn.application_args[3])),
  Return(Int(1))
])

```




Burada diyor ki, `on_creation` durumunda aşağıdaki işlemleri sırayla yap. `App.globalPut(Bytes("Creator"))` bilgisini `Txn.sender()`'a eşitle, `Assert(Txn.application_args.length()` uzunluğunu, yani alacağı argüman uzunluğunu 4 olarak belirle. Bu 4 argüman da alt satırdaki `App.globalPut(Bytes("RegBegin"))`, `App.globalPut(Bytes("RegEnd"))`, `App.globalPut(Bytes("VoteBegin"))` ve `App.globalPut(Bytes("VoteEnd"))` oluyor. Yani oylama döneminin başlangıcı ve bitişi, oylamanın başlangıcı ve oylamanın bitişi. Şöyle düşünün bir referandum yapılıyor. Cumartesi ve Pazar günü oy verme hakkınız var. Cumartesi sabah referandumun başlangıcı `RegBegin`, Pazar akşam bitiş zamanı ise `RegEnd` değişkeninde tutuluyor. Sizin oy vermenizin başlangıcı `VoteBegin`, bitişiyse `VoteEnd` değişkeninde tutuluyor.

Not: Burada `Btoi` şeklinde bir ifade geçiyor. Bunun anlamı bir byte dizisini `unit64`'e dönüştür.

Oylama başlangıç ve bitiş zamanlarını aşağıdaki kod bloğu içerisinde giriyoruz. Diyoruz ki oylamanın çalışması yani `on_register`, `RegBegin` zamanından sonra ve `RegEnd` zamanından önce olmalıdır. Bu tarihleri bir `And` kapısından geçirerek zamanı belirliyoruz.

```
on_register = Return(And(
    Global.round() >= App.globalGet(Bytes("RegBegin")),
    Global.round() <= App.globalGet(Bytes("RegEnd"))
))
```

Şimdi farklı bir bölüm ile devam edelim.

```
choice = Txn.application_args[1]
choice_tally = App.globalGet(choice)
on_vote = Seq([
    Assert(And(
        Global.round() >= App.globalGet(Bytes("VoteBegin")),
        Global.round() <= App.globalGet(Bytes("VoteEnd"))
    )),
    get_vote_of_sender,
    If(get_vote_of_sender.hasValue(),
        Return(Int(0))
    ),
    App.globalPut(choice, choice_tally + Int(1)),
    App.localPut(Int(0), Bytes("voted"), choice),
    Return(Int(1))
])
```



Bu bölüm, bir hesabın oylaması ile ilgili gövde diyebiliriz. İlk olarak, mevcut turun ve içinde olduğundan emin olmak için `on_vote` bir Assert ifade kullanır. Bu mevcudiyetten emin olunduktan sonra gönderenin hesabının yerel durumunda anahtara sahip olup olmadığı kontrol edilir. Değişken, programda daha önce yetkilendirilmiş bir alma işlemi olan olarak tanımlanmıştır. Normal alma işlemlerinin aksine, genişletilmiş sürüm, ek-sik anahtarlar için varsayılan bir 0 değeri döndürmek yerine bir anahtarın var olup olmadığını kontrol etmemizi sağlar. Hesabın yerel durumunda anahtarı varsa, zaten oy vermişlerdir ve program 0 döndürerek başarısız olur.

Aksi bir durum olduysa, yani hesap henüz oy vermediyse program, gönderenin oy vermek istediği seçeneği `Txn.application_args[1]` ile alır. Ayrıca `choice_tally`, seçimin sahip olduğu mevcut oy sayısını da alır. Kod, sayımı 1 artırır ve yeni değeri tekrar genel duruma yazar. Ardından, "voted" hesabın yerel erişim alındaki anahtara oy verdikleri seçeneği yazarak hesabın başarıyla oy kullandığını kaydeder.

Kodumuzdaki bir başka seq yani dizi işlem olan `on_closeout` a bakalım.

```
on_closeout = Seq([
    get_vote_of_sender,
    If(And(Global.round() <= App.globalGet(Bytes("VoteEnd")), get_
vote_of_sender.hasValue()),
    App.globalPut(get_vote_of_sender.value(), App.globalGet(get_
vote_of_sender.value()) - Int(1))
),
    Return(Int(1))
])
```

Bu kodda ise oyun atıldığı kontrol ediliyor ve eğer bir oy atma daha girişimi olursa artan değeri 1 azaltarak bunun önüne geçiyor. Yani bir tür hile kontrolü ve oylamanın bitirildiğini onaylama kısmı diyebiliriz.

İlk başta iki farklı metod üzerinde çalışacağımızdan bahsetmiştik. Şimdi bunlardan `clear_state` program kısmına da bakalım.



```
def clear_state_program():
    get_vote_of_sender = App.localGetEx(Int(0), App.id(), Bytes("voted"))
    program = Seq([
        get_vote_of_sender,
        If(And(Global.round() <= App.globalGet(Bytes("VoteEnd")), get_vote_of_sender.hasValue()),
            App.globalPut(get_vote_of_sender.value(), App.globalGet(get_vote_of_sender.value()) - Int(1))
        ),
        Return(Int(1))
    ])

```

Bu koda yakından bakalım.

`get_vote_of_sender = App.localGetEx(Int(0), App.id(), Bytes("voted"))` oy kullanan kişiyi `voted` olarak kaydediyor. Kodun geri kalanındaki `program =` ile başlayan kısım ise `get_vote_of_sender = App.localGetEx(Int(0), App.id(), Bytes("voted"))` oy kullanan kişiyi `voted` olarak kaydeder.

Kod bloklarımızı inceledik. Şimdi main kısmına tekrar bakalım. İki farklı çıktı alacağımızı görmüştük. Bunlar `vote_approval.teal` , `vote_clear_state.teal` dosyalarıydı.

Kodumu derlediğimde o an çalıştığımız dizinde aşağıdaki iki dosyanın oluştuğunu görüyoruz.

 <code>vote_approval.teal</code>	1 KB teal File	18.12.2020 22:23
 <code>vote_clear_state.teal</code>	252 bytes teal File	18.12.2020 22:23

İçerikler ise şu şekilde oluyor.

`Vote_approval` sözleşmesi:

```
#pragma version 2
txn ApplicationID
int 0
==
bnz l14
txn OnCompletion
int DeleteApplication
==
bnz l15
txn OnCompletion
int UpdateApplication
==

```



```
bnz l16
txn OnCompletion
int CloseOut
==
bnz l17
txn OnCompletion
int OptIn
==
bnz l18
txna ApplicationArgs 0
byte "vote"
==
bnz l19
err
l14:
byte "Creator"
txn Sender
app_global_put
txn NumAppArgs
int 4
==
bnz l21
err
l21:
byte "RegBegin"
txna ApplicationArgs 0
btoi
app_global_put
byte "RegEnd"
txna ApplicationArgs 1
btoi
app_global_put
byte "VoteBegin"
txna ApplicationArgs 2
btoi
app_global_put
byte "VoteEnd"
txna ApplicationArgs 3
btoi
app_global_put
int 1
```



```
return
b l20
l15:
txn Sender
byte "Creator"
app_global_get
==
return
b l20
l16:
txn Sender
byte "Creator"
app_global_get
==
return
b l20
l17:
int 0
global CurrentApplicationID
byte "voted"
app_local_get_ex
store 0
store 1
global Round
byte "VoteEnd"
app_global_get
<=
load 0
&&
bz l22
load 1
load 1
app_global_get
int 1
-
app_global_put
l22:
int 1
return
b l20
l18:
```



```
global Round
byte "RegBegin"
app_global_get
>=
global Round
byte "RegEnd"
app_global_get
<=
&&
return
b l20
l19:
global Round
byte "VoteBegin"
app_global_get
>=
global Round
byte "VoteEnd"
app_global_get
<=
&&
bnz l23
err
l23:
int 0
global CurrentApplicationID
byte "voted"
app_local_get_ex
store 0
store 1
load 0
bz l24
int 0
return
l24:
txna ApplicationArgs 1
txna ApplicationArgs 1
app_global_get
int 1
```



```
+
app_global_put
int 0
byte "voted"
txna ApplicationArgs 1
app_local_put
int 1
return
l20:
```

Vote_clear_state sözleşmesi;

```
#pragma version 2
int 0
global CurrentApplicationID
byte "voted"
app_local_get_ex
store 0
store 1
global Round
byte "VoteEnd"
app_global_get
<=
load 0
&&
bz l25
load 1
load 1
app_global_get
int 1
-
app_global_put
l25:
int 1
return
```

Peki neden iki farklı sözleşme oluştu tekrar hatırlayalım. Vote_approval sözleşmesi oyu kullanmayı raporluyor, diğer sözleşme olan Vote_clear_state ile oyun güvenli ve doğrulanmış olarak kabul edildiğini raporluyor.

Bu uygulama nispeten daha kompleks ama sonuçta bir referandum sis-

temini kodlamış bulunduk. Bu referandum sistemi yerel veya ulusal ölçekte insanların seçim yapmasında kullanılabilir.

Peki bu yazdığımız akıllı kontratları ağa yüklemenin en kolay yolu hangisi? Benim tavsiyem <https://teal.algodesk.io/> sitesi. Buraya giriyoruz.

Create Contract

Contract Name *

Sözleşmem|

.teal

Cancel

Create

Sözleşmeyi yükleyip Compile diyoruz ve işlem tamam!

Mainnet

```

1  #pragma version 2
2  txN Fee
3  int 1000
4  <br>
5  txN TypeEnum
6  int 1
7  <br>
8  txN RekeyTo
9  global ZeroAddress
10 <br>
11 &&
12 &&
13 txN CloseRemainderTo
14 global ZeroAddress
15 <br>
16 txN Receiver
17 addr 7ZSPW0ZC6LFNQFGHMKSKS471QP50JW2M3HA20PKTY3WTNP5NUZPHBmZ7M
18 <br>
19 &&
20 txN Amount
21 int 100000
22 <br>
23 &&
24 txN FirstValid
25 itob
26 arg 0
27 addr 7ZSPW0ZC6LFNQFGHMKSKS471QP50JW2M3HA20PKTY3WTNP5NUZPHBmZ7M
28 ed25519verify
29 &&
30 txN Lease
31 txN FirstValid
32 itob
33 sha256
34 <br>
35 &&
36 txN CloseRemainderTo
37 addr 6ZHGH5Z5CTPCF5WCESXNGRSVK7QJTR63KMNY5F3CUYDH057VTCM30BGY
        
```

Variables

Name *

\$

Value *

⬇ Add

Console

Success

Contract Address

SNDK2AGDCE3DBP9KWOLPSSLLJG3JP8R8
KUFPQ7AYTPSSLZW8Ud4

Result

A8AEACbuUGACtCP5w-tcBuYTFMygrH9ACQI
yhaZuqHqkzmlLW0oP7DYS-S6kbFTYEJZA
Bps-TLxvYnVetPndFrsm-wrChOjQuD8D
EhAZHQkyanmySLSdeUBQM2WLSgtEDEGM
QRmAnQh-qurjLLmqHqkzlvAREA==

🏃 Run Transactions

Kontratımızı kontrol edelim.

Search by Address / Tx ID / Group Tx ID / Block / Asset Name / Asset ID

MAINNET

Wallets

Assets

Statistics

Blockchain

Tools

Developer API

Algorand Account Overview

Status

Offline

Address

51KQIO24U5HCED3BI6K5WLOP5ISLJXG3P6BKUFYQ7JV7V5O...

QR

Copy

Balances

▲ 0

Rewards

▲ 0

Balance History

4

2

0

Total Transactions: 0

Transactions

TxID	Block	Age	Amount	From	To	Fee	Type
No transactions							

DÖRDÜNCÜ BÖLÜM

REACH İLE DAPP KODLAMAYA GİRİŞ

Bu bölümden itibaren geçtiğimiz aylarda çıkan ve son derece iddialı bir proje olan Reach üzerinde nasıl kodlama yapabileceğinize değineceğiz. Reach ile ilgili ilk Türkçe kaynak olması da anlamlı olacak.

Reach platformuna buradan ulaşabilirsiniz. Proje kendisini en güvenli ve kolay şekilde Dapp geliştirme platformu olarak adlandırıyor. Bir saniye duralım ve Dapp'ın ne olduğuna tekrar bakalım.

Bu soruya Wikipedia şöyle cevap veriyor.

Merkezi olmayan, dağıtılmış bir bilgi işlem sistemi üzerinde çalışan bir bilgisayar uygulamasıdır. DApp'ler, DApp'lerin genellikle akıllı sözleşmeler olarak anıldığı Ethereum Blok Zinciri gibi dağıtılmış defter teknolojileri tarafından popüler hale getirildi.

Bu tanım güzel fakat eksik. Çünkü Ethereum ile gündem olsa da artık Algorand başta olmak üzere başka akıllı kontrat sistemlerinde de Dapp gündemde. Reach projesi de Ethereum ve Algorand üzerinde geliştirme yapabilmenizi sağlıyor. Aralık 2020 bilgi notunda paylaştıklarına göre Cardano teknolojisinin arkaplanındaki Plutus'u da sisteme entegre etme sürecindeler.

Biz biliyoruz ki henüz geleneksel programlamacıların blokzincir ekosistemine “Kavimler Göçü” başlamadı. Bundan dolayı Reach tarzı uygulamaları öncü aktörler olarak görebiliriz. Bu kaçınılmaz göç başladığında, blokzincir tabanlı çözümler kulaktan kulağa yayıldığında, Reach tarzı çoklu platform desteği olan çözümler değerini daha da katlayacak.

Yine biliyoruz ki Blokzincir konusu farklı bir teknolojiye göre çoğu zaman daha karışık. Örneğin Artırılmış Gerçeklik gibi bir teknolojinin gerek mantığını gerek kodsız arkaplanını anlamak bir kaç saatinizi alırken, Blokzincir'de aynı sürede sadece kafanızda Bizanslı generalleri dolaştırmaya başlayabiliyorsunuz. Hal böyle olunca insanlar da bu teknolojiden geride duruyor. Fakat Reach tarzı platformlar bu süreci günlerden dakikalara indirebilecek çözümler sunma noktasında iddialı.



Reach nasıl çalışır?

Peki nasıl çalışır bu Reach. Sistem sizin Reach'in kendi dilinde yazdığınız kodları farklı blokzincir teknolojilerinde kullanılabilir şekilde derler.

Projeyi kim yaptı?

Projenin iki kurucusu var. Chris Swenor ve Jay Maccarty. Chris işin teknik, Jay ise teorik tarafında daha çok. Reach öğrenirken de projenin kurucularından Chris ile takıldığım noktalarda Discrod üzerinden görüşme şansı oldu. Ayrıca Youtube üzerinde de düzenli olarak eğitim videoları yayınlıyor.

Bir sonraki yazıda Reach'in Windows'a kurulumuna geçiyoruz.

Windows'a Reach Kurulumu

Windows üzerinde Reach kurulumu bir kaç adımdan oluşuyor. Eğer docker kullanmadıysanız ilk etapta karışık gibi gelse de üstesinden geliyor.

İlk olarak Windows Versiyonumuza bakıyoruz. Windows 10 işletim sistemine ve versiyonuna bakıyoruz. Bunun için Başlat'a winver yazıp Enter dediginizde karşınıza aşağıdaki gibi bir pencere açılacaktır.



Buradaki güncellik bizim için önemli. Eğer güncel değilse başlat üzerinden Windows Update ayarlarına girerek güncelleme yapabilirsiniz.

İkinci adımda Windows Subsystem for Linux veya kısa ismiyle WSL kurulumu yapıyoruz. WSL, Windows 10 üzerinde sanal makine yardımıyla bir Linux dağıtımı kurmaya gerek kalmadan bize Linux ortamı sağlayan bir özellik diyebiliriz. Bu sayede Linux'ta kullandığımız birçok komut satırı



uygulamasını kullanabiliyoruz. Kurulum adımları burada anlatılmış. Biz de özetleyelim.

Başlattan Windows PowerShell'i yönetici olarak açıyoruz. Sırasıyla aşağıdaki komutları çalıştırıyoruz.

dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart

```
PS C:\WINDOWS\system32> dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart
Deployment Image Servicing and Management tool
Version: 10.0.19041.572
Image Version: 10.0.19042.685
Enabling feature(s)
[=====100.0%=====]
The operation completed successfully.
PS C:\WINDOWS\system32>
```

Ardından WSL'yi güncel tutabilmek için Windows versiyonumuzu güncelliyoruz, aksi takdirde WSL 2 sorunsuz çalışmıyor. İlk başta güncellediyseniz devam edebilirsiniz.

Sonraki adımda sanal makineyi aşağıdaki kod ile çalışır duruma getiriyoruz.

dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart

```
PS C:\WINDOWS\system32> dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart
Deployment Image Servicing and Management tool
Version: 10.0.19041.572
Image Version: 10.0.19042.685
Enabling feature(s)
[=====100.0%=====]
The operation completed successfully.
```

Dördüncü adımda Windows için Linux çekirdeğini buradan indiriyoruz.

Beşinci adımda PowerShell'e wsl -set-default-version 2 yazarak. WSL için kullanacağımız varsayılan versiyonun WSL 1 değil WSL 2 olmasını sağlıyoruz. Altıncı adımda bir Linux paketi seçip kuruyoruz. Ben Ubuntu 20.04 seçtim.

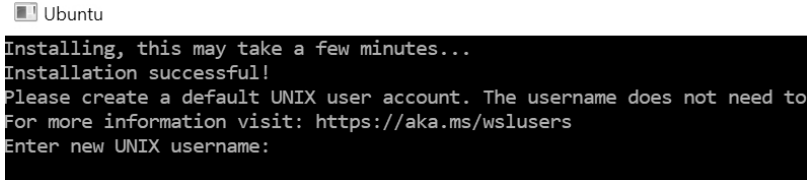
- Ubuntu 16.04 LTS
- Ubuntu 18.04 LTS
- Ubuntu 20.04 LTS
- openSUSE Leap 15.1
- SUSE Linux Enterprise Server 12 SP5
- SUSE Linux Enterprise Server 15 SP1
- Kali Linux
- Debian GNU/Linux
- Fedora Remix for WSL



- Pengwin
- Pengwin Enterprise
- Alpine WSL

Kurulum sonrasında Ubuntu'yu Başlat üzerinden bulup, çalıştırıyoruz. Bir kullanıcı adı ve şifre belirliyoruz.

```


  Installing, this may take a few minutes...
  Installation successful!
  Please create a default UNIX user account. The username does not need to
  For more information visit: https://aka.ms/wslusers
  Enter new UNIX username:

```

Ana kurulumumuzun ikinci adımı olan WSL sürecini tamamladık. Üçüncü adıma geçelim. Bu adımda Docker kuruyoruz.

Docker'ın sitesinden, Docker Desktop kurup, Docker üzerindeki WSL 2 yi aktif duruma getiriyoruz.

Son olarak da Docker ile WSL 2 bağlantısını yapıyoruz.

Şimdi Reach kurulumuna geçelim. Yukarıdaki işlemlerin ardından Ubuntu'yu veya seçtiğiniz Linux çeşidini açarak aşağıdaki kodları yazıyoruz.

```

$ make -version
$ docker -version
$ docker-compose -version

```

Bu sayede Make, Docker ve Docker-Compose'un kurulu olduğundan emin oluyoruz. Çünkü bunların birisi kurulu değilse Reach kurulamıyor.

Ardından,

```
$ mkdir -p ~/reach/tut && cd ~/reach/tut
```

komutuyla reach/tut diye bir klasör açıyoruz. Siz de istediğiniz ismi verebilirsiniz.

Bu klasörün içine,

```
$ curl https://raw.githubusercontent.com/reach-sh/reach-lang/master/reach -o reach ; chmod +x reach
```

komutuyla Github üzerinden Reach'i kuruyoruz. Eğer kurulum sorunsuz olduysa,

```
$ ./reach version
```

dediğimizde aşağıdaki şekilde 0.1 yazması gerekiyor.



```
bugra@bugraayan2-n:~$ reach --version
reach: command not found
bugra@bugraayan2-n:~$ ./reach --version
-bash: ./reach: Is a directory
bugra@bugraayan2-n:~$ ./reach version
-bash: ./reach: Is a directory
bugra@bugraayan2-n:~$ mkdir -p ~/reach/tut && cd ~/reach/tut
bugra@bugraayan2-n:~/reach/tut$ curl https://raw.githubusercontent.com/reach-sh/reach-lang
/master/reach -o reach ; chmod +x reach
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
100 25327 100 25327 0 0 35030 0 --:--:-- --:--:-- --:--:-- 35176
bugra@bugraayan2-n:~/reach/tut$ ./reach version
reach 0.1
bugra@bugraayan2-n:~/reach/tut$
```

Eğer ./reach help dersek de,

```
bugra@bugraayan2-n:~/reach/tut$ ./reach help
Usage: reach COMMAND
where COMMAND is one of
  compile --- compile an app
  clean --- delete compiled artifacts
  init --- set up source files for a simple app
  run --- run a simple app
  down --- halt any dockerized devnets for this app
  scaffold -- set up Docker scaffolding for a simple app
  upgrade --- upgrade Reach
  update --- update Reach Docker images
  version --- display version
  help --- show this info
```

aşağıdaki şekilde yardım menüsü geliyor.

Özetlemek gerekirse; Windows'u güncelledik, WSL kurduk, WSL ile koordineli çalışacak Linux dağıtımını kurduk, Docker kurduk, Docker WSL2 bağlantısını yaptık ve son olarak Reach kurduk.

Reach Üzerinde Taş Kâğıt Makas Oyunu Yapma

Taş, kâğıt, makas oyununa devam ediyoruz. Bu bölüm için de Ubuntu üzerinde tut-2 isimli bir klasör oluşturup için index.rsh dosyasını açıyoruz.

```
bugra@bugraayan2-n:~/reach/tut$ cd ..
bugra@bugraayan2-n:~/reach$ mkdir tut-2
bugra@bugraayan2-n:~/reach$ touch index.rsh
bugra@bugraayan2-n:~/reach$ nano index.rsh
```

Dosyanın içine aşağıdaki kodu giriyoruz.



```
'reach 0.1';

const Player =
  { getHand: Fun([], UInt),
    seeOutcome: Fun([UInt], Null) };

export const main =
  Reach.App(
    {},
    [['Alice', Player], ['Bob', Player]],
    (A, B) => {
      A.only() => {
        const handA = declassify(interact.getHand());
        A.publish(handA);
        commit();

        B.only() => {
          const handB = declassify(interact.getHand());
          B.publish(handB);

          const outcome = (handA + (4 - handB)) % 3;
          commit();

          each([A, B], () => {
            interact.seeOutcome(outcome);
          });
        }
      }
    }
  );
```

Bu kodda ilk örnekte olduğu gibi reach 0.1 ile başlıyoruz. Fakat burada main metodu yerine Player diye bir sınıf tanımlıyoruz. Bu oyuncunun hamleleri ile ilgili bilgileri içeriyor. Gethand kısmıyla oyuncunun hamlesini al, seeOutcome kısmıyla da bu hamlenin sonucunda ne olacağını söyle kısımlarını belirtiyoruz.

Bu metoddan sonra ana metodumuza geçiyoruz. Burada da ilk örnekten farklı olarak Alice ve Bob'un yanında dikkat ederseniz Player ifadesi var. Bu Alice ile Bob'un sınıfının Player olduğunu gösteriyor.

A.only() ve B.only() ile başlayan kısımlarda ise hamleyi alıyor ve ağa yayınlıyor.

Burada işin matematiği ise $\text{const outcome} = (\text{handA} + (4 - \text{handB})) \% 3$; kısmında oluşturulmuş. Şöyle düşünün. Bir taş, kâğıt, makas oyununda kaç sonuç ihtimali vardır? Ya birinci oyuncu ya ikinci oyuncu kazanır ve yahut berabere kalınır. Yani 3 durum vardır. İşte bunun için mod 3'e göre sonuç döndüren bir kod yazılmış.



Alice'in seçimini alıp, (4-Bob'un seçimi) ile topluyor ardından da bunun mod 3'e göre sonucuna bakıyor.

Şimdi ikinci dosyamız olan index.mjs ye geçelim. Kodumuz aşağıdaki şekilde.

```
import { loadStdlib } from '@reach-sh/stdlib';
import * as backend from './build/index.main.mjs';

(async () => {
  const stdlib = await loadStdlib();
  const startingBalance = stdlib.parseCurrency(10);

  const accAlice = await stdlib.newTestAccount(startingBalance);
  const accBob = await stdlib.newTestAccount(startingBalance);

  const ctcAlice = accAlice.deploy(backend);
  const ctcBob = accBob.attach(backend, ctcAlice.getInfo());

  const HAND = ['Rock', 'Paper', 'Scissors'];
  const OUTCOME = ['Bob wins', 'Draw', 'Alice wins'];
  const Player = (Who) => ({
    getHand: () => {
      const hand = Math.floor(Math.random() * 3);
      console.log(`${Who} played ${HAND[hand]}`);
      return hand;
    },
    seeOutcome: (outcome) => {
      console.log(`${Who} saw outcome ${OUTCOME[outcome]}`);
    },
  });

  await Promise.all([
    backend.Alice(
      ctcAlice,
      Player('Alice'),
    ),
    backend.Bob(
      ctcBob,
      Player('Bob'),
    ),
  ]);
})();
```



Ubuntu üzerinde dosyamızı oluşturup bu kodları içine koyuyoruz.

```
bugra@bugraayan2-n:~/reach$ touch index.mjs
bugra@bugraayan2-n:~/reach$ nano index.mjs
```

```
import { loadStdlib } from '@reach-sh/stdlib';
import * as backend from './build/index.main.mjs'; ile kütüphaneleri-
```

mizi içe aktardık.

```
const startingBalance = stdlib.parseCurrency(10);
```

kısımında 10 ile sabit başlangıç bakiyesi belirledik. Diğer sabitlerimiz,

```
const accAlice = await stdlib.newTestAccount(startingBalance);
const accBob = await stdlib.newTestAccount(startingBalance);

const ctcAlice = accAlice.deploy(backend);
const ctcBob = accBob.attach(backend, ctcAlice.getInfo());

const HAND = ['Rock', 'Paper', 'Scissors'];
const OUTCOME = ['Bob wins', 'Draw', 'Alice wins'];
```

ile Alice ve Bob'un başlangıç bakiyelerini, arkaplanda kullanacakları yapıları, oyundaki seçimlerin isimlerini ve sonucun hangi karakter dizileriyle döndürüleceğini belirttik.

Buradaki gethand fonksiyonu içinde Math.random içinde 0 ile 1 arası rastgele bir değer alıp onu 3 ile çarpıp, flood ile tam sayıya yuvarlama yapıyor. Oyuncularımızın rastgele seçimler yapması bu şekilde sağlanıyor. Eğer sonuç 0 döndürürse Rock, 1 döndürürse Paper 2 döndürürse Scissors anlamına geliyor.

```
console.log(`${Who} played ${HAND[hand]}`);
```

ile hamle kayıt altına alınıyor.

```
console.log(`${Who} saw outcome ${OUTCOME[outcome]}`);
```

ile de sonuç loglanıyor.

Son olarak ilk örnekte olduğu gibi backend kullanımlarımızı gösterip kodumuzu bitiriyoruz.

Bu iki dosyayı ./reach run diyerek çalıştırıyoruz.



```
=> [2/3] COPY . /app 0.0s
=> [3/3] RUN cp /app/package.json.index /app/package.json 0.3s
=> exporting to image 0.0s
=> => exporting layers 0.0s
=> => writing image sha256:7347752d2f8942cc42d4362918f5df807b8db8a98dccc 0.0s
=> => naming to docker.io/reachsh/reach-app-tut-index:latest 0.0s
/Users/enespolat/reach/tut/reach run index
docker build -f Dockerfile.index --tag=reachsh/reach-app-tut-index:latest .
[+] Building 0.1s (8/8) FINISHED
=> [internal] load build definition from Dockerfile.index 0.0s
=> => transferring dockerfile: 261B 0.0s
=> [internal] load .dockerignore 0.0s
=> => transferring context: 34B 0.0s
=> [internal] load metadata for docker.io/reachsh/runner:0.1 0.0s
=> [internal] load build context 0.0s
=> => transferring context: 3.66kB 0.0s
=> [1/3] FROM docker.io/reachsh/runner:0.1 0.0s
=> CACHED [2/3] COPY . /app 0.0s
=> CACHED [3/3] RUN cp /app/package.json.index /app/package.json 0.0s
=> exporting to image 0.0s
=> => exporting layers 0.0s
=> => writing image sha256:7347752d2f8942cc42d4362918f5df807b8db8a98dccc 0.0s
=> => naming to docker.io/reachsh/reach-app-tut-index:latest 0.0s
docker-compose -f "docker-compose.yml.index" run --rm reach-app-tut-index-${REACH_H_CONNECTOR_MODE} index
WARNING: Found orphan containers (tut_ganache_1) for this project. If you remove
d or renamed this service in your compose file, you can run this command with th
e --remove-orphans flag to clean it up.
Creating tut_reach-app-tut-index-ETH-test-dockerized-geth_run ... done

> @reach-sh/tut-index@ index /app
> node --experimental-modules --unhandled-rejections=strict index.mjs

(node:19) ExperimentalWarning: The ESM module loader is experimental.
Alice played Rock
Bob played Paper
Alice saw outcome Bob wins
Bob saw outcome Bob wins
```

Oyunumuz oynandı. Alic Scissors yani makas, Bob Rock yani taş seçti. Dolayısıyla oyunu Bob kazandı.

Kodu ./reach run ile her çalıştırdığımızda farklı bir sonuç dönebileceğini görüyoruz.

Reach Üzerinde Taş Kâğıt Makas Oyunu Yapma

Bu bölümde Reach üzerinde bir taş kâğıt makas oyunu geliştireceğiz. Bu sayede hem Reach'i kullanmayı göreceğiz hem de bir oyun geliştirmiş olacağız.

Bir önceki bölümde Reach'in versiyonunu kontrol etmiş ve help menüsüyle Reach ile ilgili bilgileri almıştık. Uygulamamızı geliştirmeye başlamadan önce Ubuntu üzerinden,

./reach update diyerek imajların docker içerisine yüklenmesini sağlıyoruz. Bu işlem yaklaşık 30 dakika sürüyor. Yükleme sonrasında tekrar aynı komutu çalıştırdığımızda aşağıdaki ekran görüntüsünü alıyoruz.



```
bugra@bugraayan2-n:~/reach/tut$ ./reach update
0.1: Pulling from reachsh/reach
Digest: sha256:e3c169f89df7d4db30901f9c8a682c63bd540fd149052cb9426a172ec214c82a
Status: Image is up to date for reachsh/reach:0.1
docker.io/reachsh/reach:0.1
0.1: Pulling from reachsh/stdlib
Digest: sha256:47b33d533b4493ff40ed196b453081cc8752c45ea692e58e6abb097149905380
Status: Image is up to date for reachsh/stdlib:0.1
docker.io/reachsh/stdlib:0.1
0.1: Pulling from reachsh/runner
Digest: sha256:375c8822b701be880315046f814531f7c6a8e55cb0a1c229fbb0933ddfdb6072
Status: Image is up to date for reachsh/runner:0.1
docker.io/reachsh/runner:0.1
```

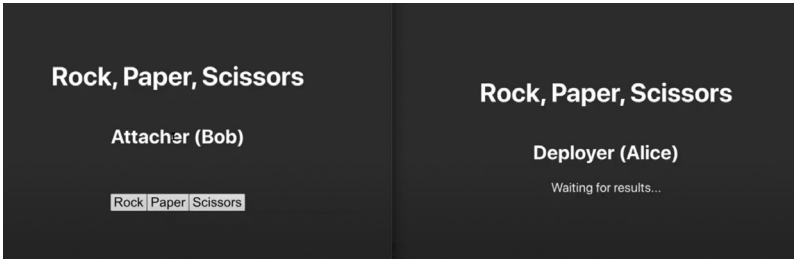
Burada hem Algorand hem de Ethereum geliştirici ağları devnetler yükleniyor.

Docker üzerindeki görünüm de aşağıdaki şekilde oluyor.

Images on disk				
45 images Total size: 9.41 GB				
IN USE UNUSED Clean up...				
LOCAL REMOTE REPOSITORIES				
reachsh/reach	0.1	356b0181dc2b	1 day ago	236.13 MB
reachsh/react-runner	0.1	b2178946e15a	1 day ago	1.27 GB
reachsh/rpc-server	0.1	f77642e00ca9	1 day ago	1.11 GB
reachsh/runner	0.1	1dd1b27dbcee	1 day ago	1.11 GB
reachsh/algorand-devnet	0.1	a81e4356db1a	1 day ago	6.26 GB
reachsh/ethereum-devnet	0.1	040ad8c74684	1 day ago	45.69 MB
<none>	<none>	f688b75ea83f	about 1 month ago	1.53 GB
reachsh/reach	<none>	8eb1ae9fe68a	about 1 month ago	237.42 MB
reachsh/react_runner	0.1	24a14ec70d3d	about 1 month ago	1.25 GB
reachsh/ethereum-devnet	<none>	0cd702e879b9	about 1 month ago	45.69 MB
reachsh/algorand-devnet	<none>	035c54d16b77	about 1 month ago	6.26 GB
reachsh/runner	<none>	1f4e8076fc72	about 1 month ago	1.53 GB
reachsh/stdlib	0.1	2f0d6728dc0c	about 1 month ago	1.53 GB
reachsh/reach	<none>	1e67af8dd73e	about 1 month ago	237.38 MB

Artık uygulamamızı kodlamaya geçebiliriz. Bu eğitimde, Rock, Paper, Scissors'ın bir versiyonunu oluşturacağız. Burada iki oyuncu, Alice ve Bob, oyunun sonucuna göre puan kazanabili. Basit adımlarla başlayıp yavaş yavaş uygulamayı daha detaylı hale getireceğiz.

Uygulama ekranımız aşağıdaki şekilde oluyor. Blokzincir uygulamalarının standartlaşmış kişileri Alice ve Bob bir oyun oynuyor. Sol tarafta Bob bir seçim yapıyor, ardından Alice onun seçimini tahmin ederek kendi seçimini yapıyor ve sonuca göre bir kazanan oluyor.



Kodu yazarken iki tane dosyaya ihtiyacımız olacak. Bunlar index.rsh ve index.mjs. Bu dosya uzantılarına aşina olmamanız çok normal, bunu dert etmeyin. Rsh uzantılı dosyamız arkaplanda çalışıyor, mjs uzantılı dosya ise web sitesi arayüzünü oluşturan dosyamız. Buradaki m-js , javascriptten geliyor.

Öncelikle kodlarımızı satır satır inceleyelim ardından Ubuntu üzerinden kurulumu yapalım.

Index.rsh dosyası:

```
'reach 0.1';

export const main =
  Reach.App(
    {},
    [['Alice', {}], ['Bob', {}]],
    (A, B) => {
      exit(); });
```

Kodumuz reach 0.1 ile başlıyor. Çünkü biz Reach'in ilk versiyonu olan 0.1 versiyonunda çalışıyoruz.

export const main kısmı Reach programlarının başlangıcı için kullanılan ana metod.

[['Alice', {}], ['Bob', {}]] ile uygulamanın katılımcılarını tanımlıyoruz.

(A, B) => { //Erişim tanımlayıcılarını (Ave B) bu katılımcılara bağlar ve programın gövdesini tanımlar.

Exit(); kısmıyla işlemi bitiriyoruz.

Şimdi index.mjs dosyamıza bakalım.



```
import { loadStdlib } from '@reach-sh/stdlib';
import * as backend from './build/index.main.mjs';

(async () => {
  const stdlib = await loadStdlib();
  const startingBalance = stdlib.parseCurrency(10);

  const accAlice = await stdlib.newTestAccount(startingBalance);
  const accBob = await stdlib.newTestAccount(startingBalance);

  const ctcAlice = accAlice.deploy(backend);
  const ctcBob = accBob.attach(backend, ctcAlice.getInfo());

  await Promise.all([
    backend.Alice(
      ctcAlice,
      {},
    ),
    backend.Bob(
      ctcBob,
      {},
    ),
  ]);
})(); // <-- Don't forget these!
```

Mjs dosyamızı satır satır inceleyelim.

`import { loadStdlib } from '@reach-sh/stdlib';`//Reach standart kitaplık yükleyicisini içe aktarır.

`import * as backend from './build/index.main.mjs';`//backend kısmı içe aktarır.

`(async () => {` //arayüzün gövdesi olacak asenkron fonksiyonu oluşturur.
`const stdlib = await loadStdlib();`//REACH_CONNECTOR_MODE ortam değişkenlerini yükler.

`const startingBalance = stdlib.parseCurrency(10);`//her test hesabı için başlangıç bakiyesi olarak bir miktar ağ jetonunu tanımlar .

`const accAlice = await stdlib.newTestAccount(startingBalance);` //

`const accBob = await stdlib.newTestAccount(startingBalance);`//Alice ve Bob için ilk donanımına sahip test hesapları oluşturur. Bu, yalnızca Reach tarafından sağlanan geliştirici test ağında çalışır.

`const ctcAlice = accAlice.deploy(backend);` //Alice uygulamayı konuşturandır.

`const ctcBob = accBob.attach(backend, ctcAlice.getInfo());` //Bob ile iliştilir.



```
await Promise.all([ //Tüm backendlerin tamamlanmasını bekler.
  backend.Alice(
    ctcAlice,
    {},
  ),//Alice'in backendini başlatır.
  backend.Bob(
    ctcBob,
    {},
  ),//Bob'un backendini başlatır.
]);
})(); // <-- Bunu unutmayın! //Tanımladığımız tüm asenkron fonksiyon-
ları çağırır.
```

Şimdi bu iki dosyayı Ubuntu'da nasıl kuracağımıza bakalım.

Touch komutuyla bu iki dosyayı oluşturuyoruz.

```
bugra@bugraayan2-n:~/reach/tut$ touch index.mjs index.rsh
```

Nano komutuyla dosyaları açıp düzenliyoruz.

```
bugra@bugraayan2-n:~/reach/tut$ nano index.rsh
```

```
GNU nano 4.8 index.rsh
'reach 0.1';

export const main =
  Reach.App(
    {},
    [['Alice', {}], ['Bob', {}]],
    (A, B) => {
      exit(); });
```

Ctrl O ile kaydedip Ctrl X ile çıkıyoruz.

```
GNU nano 4.8 index.mjs
import { loadStdlib } from '@reach-sh/stdlib';
import * as backend from './build/index.main.mjs';

(async () => {
  const stdlib = await loadStdlib();
  const startingBalance = stdlib.parseCurrency(10);

  const accAlice = await stdlib.newTestAccount(startingBalance);
  const accBob = await stdlib.newTestAccount(startingBalance);

  const ctcAlice = accAlice.deploy(backend);
  const ctcBob = accBob.attach(backend, ctcAlice.getInfo());

  await Promise.all([
    backend.Alice(
      ctcAlice,
      {},
```



Dosyalarımızı kaydettik. Son olarak,
./reach init ile kodlarımızı yazıyoruz.

```
enespolat@MacBook-Pro tut % cd tut-1
enespolat@MacBook-Pro tut-1 % ls
index.mjs      index.rsh
enespolat@MacBook-Pro tut-1 % cd ..
enespolat@MacBook-Pro tut % ls
reach  tut-1
enespolat@MacBook-Pro tut % ./reach init
writing index.rsh
writing index.mjs
enespolat@MacBook-Pro tut % ls
index.mjs      index.rsh      reach          tut-1
```

Son olarak ./reach run ile kodumuzu çalıştırıyoruz.

```
/Users/enespolat/reach/tut/reach run index
docker build -f Dockerfile.index --tag=reachsh/reach-app-tut-index:latest .
[+] Building 0.1s (8/8) FINISHED
=> [internal] load build definition from Dockerfile.index      0.0s
=> => transferring dockerfile: 261B                            0.0s
=> [internal] load .dockerignore                             0.0s
=> => transferring context: 34B                                  0.0s
=> [internal] load metadata for docker.io/reachsh/runner:0.1  0.0s
=> [internal] load build context                             0.0s
=> => transferring context: 3.66kB                             0.0s
=> [1/3] FROM docker.io/reachsh/runner:0.1                  0.0s
=> CACHED [2/3] COPY . /app                                  0.0s
=> CACHED [3/3] RUN cp /app/package.json.index /app/package.json 0.0s
=> exporting to image                                         0.0s
=> => exporting layers                                          0.0s
=> => writing image sha256:faeb6afcbcac79c108b7d2b0edad18669d715ea4cceb4 0.0s
=> => naming to docker.io/reachsh/reach-app-tut-index:latest 0.0s
docker-compose -f "docker-compose.yml.index" run --rm reach-app-tut-index-$(REAC
H_CONNECTOR_MODE) index
WARNING: Found orphan containers (tut_ganache_1) for this project. If you remove
d or renamed this service in your compose file, you can run this command with th
e --remove-orphans flag to clean it up.
Creating tut_reach-app-tut-index-ETH-test-dockerized-geth_run ... done

> @reach-sh/tut-index@ index /app
> node --experimental-modules --unhandled-rejections=strict index.mjs

(node:19) ExperimentalWarning: The ESM module loader is experimental.
Hello, Alice and Bob!
```

Kurulumumuz sorunsuz olursa “Hello Alie and Bob” yazılı kısım ile komutumuz çalışıyor.

Taş Kâğıt Makas Oyununa Puan Sistemini Ekleme

Bir önceki bölümde taş kâğıt makas oyununu geliştirip, birkaç deneme yapmıştık. Bu denemelerde kazanma kaybetme durumları oluyordu fakat bir puan yoktu. Bu bölümde oyunumuza puan sistemini de dâhil ediyoruz.



Diğer Reach örneklerinde olduğu gibi bir klasör oluşturuyoruz. Bu sefer ismi tut-3 oluyor. Ardından içine index.rsh ve index.mjs dosyalarını atıyoruz.

```
tut-3 --
enespolat@MacBook-Pro tut % cd tut-3
enespolat@MacBook-Pro tut-3 % ls
index.mjs      index.rsh
enespolat@MacBook-Pro tut-3 %
```

Index.rsh dosyamızın içeriği şu şekilde:

```
'reach 0.1';

const Player =
  { getHand: Fun([], UInt),
    seeOutcome: Fun([UInt], Null) };
const Alice =
  { ...Player,
    wager: UInt };
const Bob =
  { ...Player,
    acceptWager: Fun([UInt], Null) };
export const main =
  Reach.App(
    {},
    [['Alice', Alice], ['Bob', Bob]],
    (A, B) => {
      A.only() => {
        const wager = declassify(interact.wager);
        const handA = declassify(interact.getHand()); };
      A.publish(wager, handA)
        .pay(wager);
      commit();

      B.only() => {
        interact.acceptWager(wager);
        const handB = declassify(interact.getHand()); };
      B.publish(handB)
        .pay(wager);
```



```
const outcome = (handA + (4 - handB)) % 3;
const [forA, forB] =
  outcome == 2 ? [2, 0] :
  outcome == 0 ? [0, 2] :
  [1, 1];
transfer(forA * wager).to(A);
transfer(forB * wager).to(B);
commit();

each([A, B], () => {
  interact.seeOutcome(outcome); });
exit(); });
```

Index.mjs dosyamızın içeriği ise aşağıdaki şekilde oluyor.

```
import { loadStdlib } from '@reach-sh/stdlib';
import * as backend from './build/index.main.mjs';

(async () => {
  const stdlib = await loadStdlib();
  const startingBalance = stdlib.parseCurrency(10);
  const accAlice = await stdlib.newTestAccount(startingBalance);
  const accBob = await stdlib.newTestAccount(startingBalance);
  const fmt = (x) => stdlib.formatCurrency(x, 4);
  const getBalance = async (who) => fmt(await stdlib.balanceOf(who));
  const beforeAlice = await getBalance(accAlice);
  const beforeBob = await getBalance(accBob);

  const ctcAlice = accAlice.deploy(backend);
  const ctcBob = accBob.attach(backend, ctcAlice.getInfo());

  const HAND = ['Rock', 'Paper', 'Scissors'];
  const OUTCOME = ['Bob wins', 'Draw', 'Alice wins'];
  const Player = (Who) => ({
    getHand: () => {
      const hand = Math.floor(Math.random() * 3);
      console.log(`${Who} played ${HAND[hand]}`);
      return hand;
    },
    seeOutcome: (outcome) => {
      console.log(`${Who} saw outcome ${OUTCOME[outcome]}`);
    },
  });
});
```




```
await Promise.all([
  backend.Alice(ctcAlice, {
    ...Player('Alice'),
    wager: stdlib.parseCurrency(5),
  }),
  backend.Bob(ctcBob, {
    ...Player('Bob'),
    acceptWager: (amt) => {
      console.log(`Bob accepts the wager of ${fmt(amt)}.`);
    },
  }),
]);

const afterAlice = await getBalance(accAlice);
const afterBob = await getBalance(accBob);

console.log(`Alice went from ${beforeAlice} to ${afterAlice}.`);
console.log(`Bob went from ${beforeBob} to ${afterBob}.`);

})();
```

Şimdi kodları inceleyelim. Index.rsh dosyasını inceliyoruz ama bilmemiz gerekiyor ki interaktif bir şekilde Index.mjs dosyasıyla iletişime geçeceği için kodun bazı bölümlerinde bu dosyaya yönelik kod blokları kullanıyorum.

```
const Alice =
  { ...Player,
    wager: UInt };
const Bob =
  { ...Player,
    acceptWager: Fun([UInt], Null) };
```

Kısmında wager şeklinde verilen değişken iddia değişkeni. Yani Alice veya Bob ne kadarlık bir iddiaya girecek.

Yine kodumuzda dikkat edilirse A.only ile B.only ile ifade edilen tek taraflı işlemler için yani Alice ve Bob için yazılan işlemler için birinde interact.wager diğerinde ise acceptWager(wager) geçiyor. Bunun anlamı şu Alice iddiayı oluşturuyor, Bob ise bunu kabul ediyor. Ardından da kazanması ya da kaybetmesine göre puanı azalıyor ya da artıyor.



Only kısımların altında A ve B için publish kısımları var. Bunlar da ağa yayın yapıyor. A.publish, A.only içerisinde oluşturulan handA değişkenini, B.publish de B.only içerisinde oluşturulan handB değişkenini parametre olarak alıyor.

$\text{const outcome} = (\text{handA} + (4 - \text{handB})) \% 3$; kısmı önceki Reach uygulamalarından hatırlayacağımız şekilde 0,1,2 olasılıklarını döndüren, taş kağıt makas oyunu için yapılmış bir matematiksel formül.

Şimdi gelelim bu uygulamanın en can alıcı noktasına. Aşağıdaki koda dikkatlice bakalım.

```
const [forA, forB] =
  outcome == 2 ? [2, 0] :
  outcome == 0 ? [0, 2] :
  [1, 1];
transfer(forA * wager).to(A);
transfer(forB * wager).to(B);
```

Burada mealen diyor ki eğer sonuç 2 gelirse A'ya 2 B'ye 0 gönder. Eğer sonuç 0 gelirse bunun tersini yap ve A'ya 0, B'ye 2 gönder. Diğer durumlarda yani beraberlikte ikisine de 1 ve 1 puan gönder.

Ardından da iddia oranıyla bu katsayıyı çarparak kazanan kişiye puanı gönderme işlemi yapıyor.

```
each([A, B], () => {
  interact.seeOutcome(outcome); });
exit(); });
```

SeeOutcome metoduyla da bunu ekranda gösteriyoruz.

Şimdi geçelim mjs dosyamıza.

```
const stdlib = await loadStdlib();
const startingBalance = stdlib.parseCurrency(10);
const accAlice = await stdlib.newTestAccount(startingBalance);
const accBob = await stdlib.newTestAccount(startingBalance);
```

Bu kısımlar sabit şekilde geliyor. Başlangıçta 10 birim puan başlangıç bakiyesi olarak tanımlanıp, bu puan her iki oyuncunun hesabıyla eşleştiriliyor.

Ardından, $\text{const fmt} = (x) \Rightarrow \text{stdlib.formatCurrency}(x, 4)$; buradaki 4 sayısı bir küsürat bilgisi. Yani puan aktarımlarında, puanın virgülden son-



raki 4 hanesine kadar inilebiliyor. Blokzincir üzerindeki işlemlerde mikro işlemler denilen çok küçük tutarlar bile önemli olduğu için oyun örneğinde böyle bir detaya girilmiş.

Gelelim bakiye dengeleme işlemine.

```
const getBalance = async (who) => fmt(await stdlib.balanceOf(who));
const beforeAlice = await getBalance(accAlice);
const beforeBob = await getBalance(accBob);
```

Burada asenkron bir şekilde Alice ve Bob'un bakiyesinin oluşması sağlanıyor.

Oyunu başlatan Alice sürdüren Bob olduğu için aşağıdaki kod bloğuyla bu süreci hazırlıyoruz.

```
const ctcAlice = accAlice.deploy(backend);
const ctcBob = accBob.attach(backend, ctcAlice.getInfo());
```

Şimdi bu uygulamaya özel olan wager kısmına geçelim. İlk olarak, Alice kısmının içinde geçen wager: stdlib.parseCurrency(5) kodunu görüyoruz. Burada kaybeden kişiden kazanan kişiye 5 puan geçeceğini söylüyor. Buna karşılık Bob kısmında da buna karşılık belirtilen aşağıdaki kodla da bu iddianın kabul edildiği gözüküyor.

```
acceptWager: (amt) => {
  console.log(`Bob accepts the wager of ${fmt(amt)}.`);
```

Yani oyun başında her oyuncunun 10 puanı var. Oyunu kaybeden kişinin 5 puanı, oyunu kazanan kişiye gidecek.

Kodlarımız bu şekilde. Tekrar Ubuntu'ya dönelim ve kodumuzu ./reach run ile çalıştıralım.

```
each([A, B], () => {
  interact.seeOutcome(outcome); });
exit(); });
enespolat@MacBook-Pro tut % ls
index.mjs      reach          tut-2
index.rsh      tut-1          tut-3
enespolat@MacBook-Pro tut % ./reach run
```



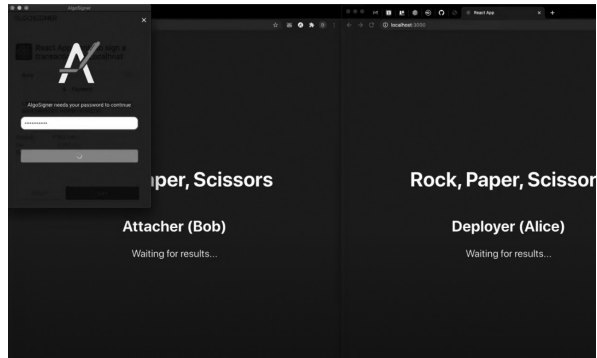
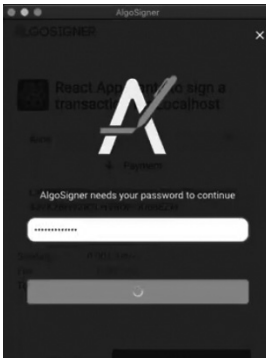
Sonuçlar aşağıdaki şekilde.

```
(node:18) ExperimentalWarning: The ESM module loader is experimental.
Alice played Paper
Bob accepts the wager of 5.
Bob played Rock
Alice saw outcome Alice wins
Bob saw outcome Alice wins
Alice went from 10 to 14.9999.
Bob went from 10 to 4.9999.
```

Alice oyunu başlattı. Paper dedi. Ardından Alice bu seçimi için 5 puanlık bir iddiaya girdi ve Bob bunu kabul etti. Ardından da Rock seçimini yaparak Alice'e yenildi. Bundan dolayı da Alice'in puanı 10'dan 14.9999 a, Bob'un puanı ise 10'dan 4.9999 a indi. Buradaki 0.0001 nereye gitti dersiniz işte bu da bu akıllı kontratı çalıştıran kişinin aldığı komisyon.

Çünkü oyun puanları Algorand üzerindeki Algorand Standard Assets'e girdiği için geliştirici bu komisyonları alarak, ASA bazında varlık sahibi olabiliyor.

Tekrar çalıştırdığımızda bu sefer farklı sonuçlar alabiliyoruz. Fakat burada bir detay dikkat çekici. Alice'in kazancı Bob'a göre daha düşük oluyor. Tabi ki bu çok küçük bir küsürat bazında. Fakat bu farkın neden olduğuna değinelim. Bunun sebebi ağa açılım işlemini Alice'in yapması. Bu onun kazancının çok küçük bir miktarda daha az olmasına sebep oluyor. Bu şekilde Dapp uygulamamızın da sonuna geldik. Bu uygulamaya React Native ile bir arayüz hazırlayarak Localhost üzerinde test edebiliyoruz. Algo Signer uygulamasıyla giriş yaparak Algo transferleri gerçekleştirilebiliyor. React Native kısmına videolarda yer verdik.



SIK SORULAN SORULAR

Bu bölümde de Algorand'a sık gelen sorular ve Algorand'ın cevaplarına yer verdik.

S1: Kötü niyetli kullanıcılar toplam hissenin $1 / 3$ 'ünden fazlasını kontrol etmediği sürece Algorand'ın fikir birliği protokolü güvenlidir. Tüm ağın $1/3$ üne sahip bir ağ, farklı bir blok oluşturabilir mi?

C: Hayır. Sürekli çalışan bir sistemde, geçmişte gizli anahtarların gelecekteki tehlikelerine karşı korumak, literatürde genellikle ileri güvenlik olarak adlandırılır. Algorand'ın fikir birliği protokolü, geçici oylama anahtarları kullanarak ileri güvenlik sağlar. Protokoldeki her oylama mesajı, işi bittikten sonra silinen bir oylama anahtarı, dolayısıyla "geçici" adı kullanılarak imzalanır. Bu yöntemle, gelecekte bir saldırgan daha önceki bir turda sistemde bulunan tüm kullanıcıları bozmayı başarsa bile, bu kullanıcılar artık daha önce üretilenlerden farklı herhangi bir blok oluşturmak için gerekli oylama anahtarlarına sahip olmayacaktır.

S2: Bir rakip, geçici anahtarlarını silmemeleri için kullanıcılara "rüşvet vererek" Algorand'ın mutabakat protokolünü bozabilir mi?

C: Hayır. Protokol kurallarına uymayı planlayan ve gerçekten de uyan bir kullanıcı dürüştür. Dürüst kullanıcılar böylece geçici anahtarlarını silecektir. Bunu yapmayan bir kullanıcı, kendisine "rüşvet verilmiş" olsun veya olmasın, kötü niyetli olur. Herhangi bir zamanda, dürüst kullanıcılar müşterek olarak mevcut toplam oy hissesinin $2 / 3$ 'ünden fazlasını elinde bulundurduğu sürece, Algorand'ın fikir birliği protokolü tamamen güvenlidir. Dahası, henüz tanımadığınız kişilere rüşvet vermek zordur ve Algorand protokolünde oy veren kullanıcılar, bireysel ve kriptografik olarak adil bir piyango aracılığıyla gizlice kendilerini seçerler.

Son olarak, büyük ölçekte gerçekleştirilen bir "rüşvet saldırısı", esasen paydaşların çoğunun sistemin güvenliği konusunda hiçbir değeri olmadığını ve ucuza satın alınabileceğini varsaydığından, gerçekçi değildir. Ger-



çekte ise tam tersi olur. Yani çoğu insan, sisteme karşı çalışmak için kendilerine rüşvet verilirse, sistemdeki paylarının tüm değerini kaybedeceklerini bilir. Bu nedenle, toplam hissenin $1/3$ 'üne rüşvet vermek, sistemin toplam değerinin en az $1/3$ 'üne mal olacaktır, bu da başarılması maliyetli bir görevdir.

S3: Algorand'ın güvenlik ve ölçeklenebilirlik ilgili garantisi nedir?

C: Algorand'ın güvenlik garantisi, toplam hissenin $2/3$ 'ü dürüst ellerde olduğu sürece hiçbir çatalın ortaya çıkamayacağıdır. Bu garanti, düşmanın sistemdeki mesaj teslimi üzerinde tam kontrole sahip olduğu ağın uzun bir bölümü sırasında da geçerlidir. Buna göre, bir blok oluşturulduktan sonra, derhal kesindir ve ağ bölümlenmiş olsun ya da olmasın blok zincirinden kaybolmayacaktır.

Algorand'ın ölçeklenebilirliğinin garantisi, ağ bölümlenmediğinde, birkaç saniyede bir blokların oluşturulması ve sonuçlandırılmasıdır. Bir ağ bölümü sırasında hiçbir blok zinciri hem güvenliği hem de ölçeklenebilirliği garanti edemez. Algorand blockchain, güvenliği her zaman garanti eder ve ağ bölümü çözüldükten sonra otomatik olarak ölçeklenebilirliği kurtarır.

S4: Hangisi daha önemli, güvenlik mi ölçeklenebilirlik mi?

C: Hem güvenlik hem de canlılık bir blockchain için çok önemlidir. Canlılık olmadan güvenliğe ulaşmak önemsiz ve anlamsızdır. Kullanıcıların hiçbir şey yapmadığı ve hiçbir blok oluşturulmadığı bir "protokol" tamamen güvenlidir, ancak ölçeklenebilirlik sağlamaz.

Örneğin, Algorand'ın aşağıdaki aşırı basitleştirmesini düşünün: Adım 1 ve Adım 2 her zamanki gibidir; Adım 3'te, her kullanıcı, aday blok B için Adım 2'den beklenen imza sayısının $2/3$ 'ünden fazlasını alıp almadığını kontrol eder. Eğer öyleyse, o tur için blok olarak B'yi işaretler. Bu kadar. Böyle bir "protokol" gerçekten güvenlidir, ancak herhangi bir ölçeklenebilirliği garanti etmez. Aslında, bir turda kötü niyetli bir blok önericisi olduğunda durur.

S5: Algorand'ın mevcut ödül planı (stake planı) nedir?

C: Algorand Vakfı, Algorand platformunun yerel simgesi olan Algos'un dağıtımından sorumludur. Her kullanıcı, Algorand topluluğunun eşit bir üyesidir ve zincire taahhüt edilen her blok için hisseleriyle orantılı bir miktar ödül kazanır. Bu ilk yaklaşım, ekosisteme erken dönemlerde geniş katılımı teşvik etmek ve Algorand topluluğundaki tüm kullanıcıları ödüllendirmek ve ademi merkeziyetçilik yolumuzu hızlandırmak için tasarlanmıştır.



Algorand'da, "teşvik" yerine "ödül" terimini kullanmayı tercih ediyoruz. Teşvik teknik bir ekonomik terim iken, bir ödül daha günlük bir ifadedir ve Algorand Vakfı'nın başlangıçta yapmayı planladığı gibi bunu herkese verseniz bile mantıklıdır. Gelecekte vakıf, uzun vadeli altyapı desteği ve sağlığı sağlamak için ek teşvik mekanizmaları getirecek.

S6: Yukarıda belirtilen ödüller nereden geliyor?

C: İlk beş yıl boyunca ödüller, Algorand Vakfı tarafından bu amaç için ayrılan ayrı bir token havuzundan gelecektir. Bundan sonra, ödüller birikmiş işlem ücretlerinden gelecektir.

S7: Ödüller hangi sıklıkta dağıtılır?

C: Tüm kullanıcılar için ödüller, blok zinciri büyüdükçe sanal olarak birikir. Ancak, hesap bir işleme dâhil olduğunda (örneğin, hesap Algos gönderdiğinde veya aldığında) bir hesabın bakiyesine açıkça yansıtılır.

S8: Algorand Vakfı, token fiyatını belirlemek için neden Hollanda açık artırmalarını kullanıyor?

C: Hollanda müzayedesini üç ana fayda sağlar: adalet, şeffaflık ve rahatlık.

Hollanda müzayedesini, piyasanın belirli bir kuruluş tarafından belirlenen fiyattan ziyade tokenlerin adil fiyatını belirlemesine izin verir. Ayrıca, bir Hollanda müzayedesinde, token fiyatı, katılımcılara adil davranarak, herhangi bir miktarda token kazanan tüm katılımcılar için aynıdır.

Hollanda müzayedesini, kullanıcıların uzaktan katılmasını kolaylaştırır. Böyle bir müzayede sırasında kullanıcıların sürekli olarak çevrimiçi kalmasına gerek yoktur. Bir teklif verebilir ve daha sonra çevrimdışı olabilir ve hatta daha sonra başka bir teklif vermek için çevrimiçi olarak geri dönebilirler.

Son olarak, şeffaflık için Algorand blok zincirinde Algo müzayedeleri yapılır. Tüm teklifler blok zincirine yerleştirilir, böylece herkes açık artırmanın düzgün bir şekilde yapıldığını doğrulayabilir.

